PHILCO

COMPUTER
DIVISION
Willow Grove, Pa.

# ELECTRONIC DATA PROCESSING SYSTEMS

## PHILCO 2000

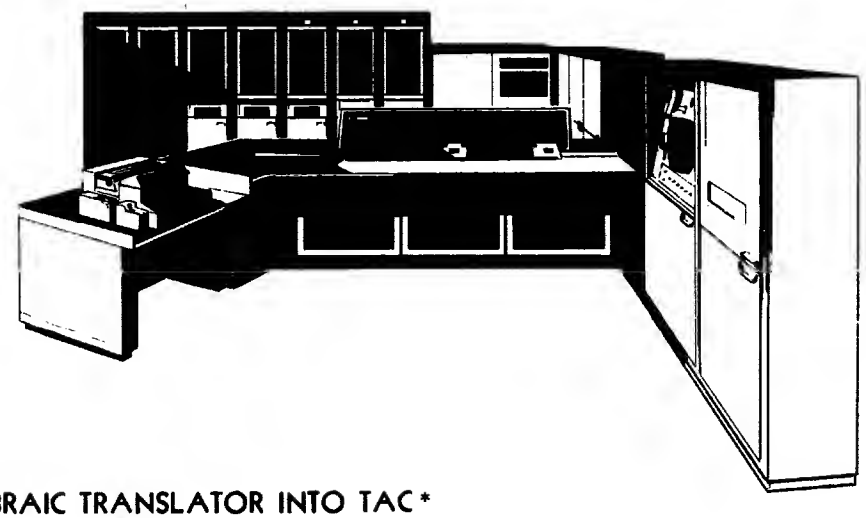
# ALTAC

## MANUAL

TM-5C

PHILCO 2000

# ALTAC*



ALGEBRAIC TRANSLATOR INTO TAC*


**PHILCO CORPORATION**

A SUBSIDIARY OF Ford Motor Company

COMPUTER DIVISION

3900 WELSH ROAD

WILLOW GROVE, PA.


* Tradename of Philco Corporation

TM-5C(2-62)

# PREFACE

ALTAC is a Philco 2000 mathematical language compiler which operates in conjunction with the TAC System to provide a running machine language program from a program written in algebraic form.

The programmer should be familiar with the Philco 2000 System and TAC coding before attempting to learn ALTAC. However, no previous knowledge of the ALTAC language or that found in other automatic coding systems is assumed. Each new concept should be thoroughly mastered in the order of presentation. A number of illustrative examples are included in each chapter. In addition, numerous exercises are presented in Appendix A.

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont'd)

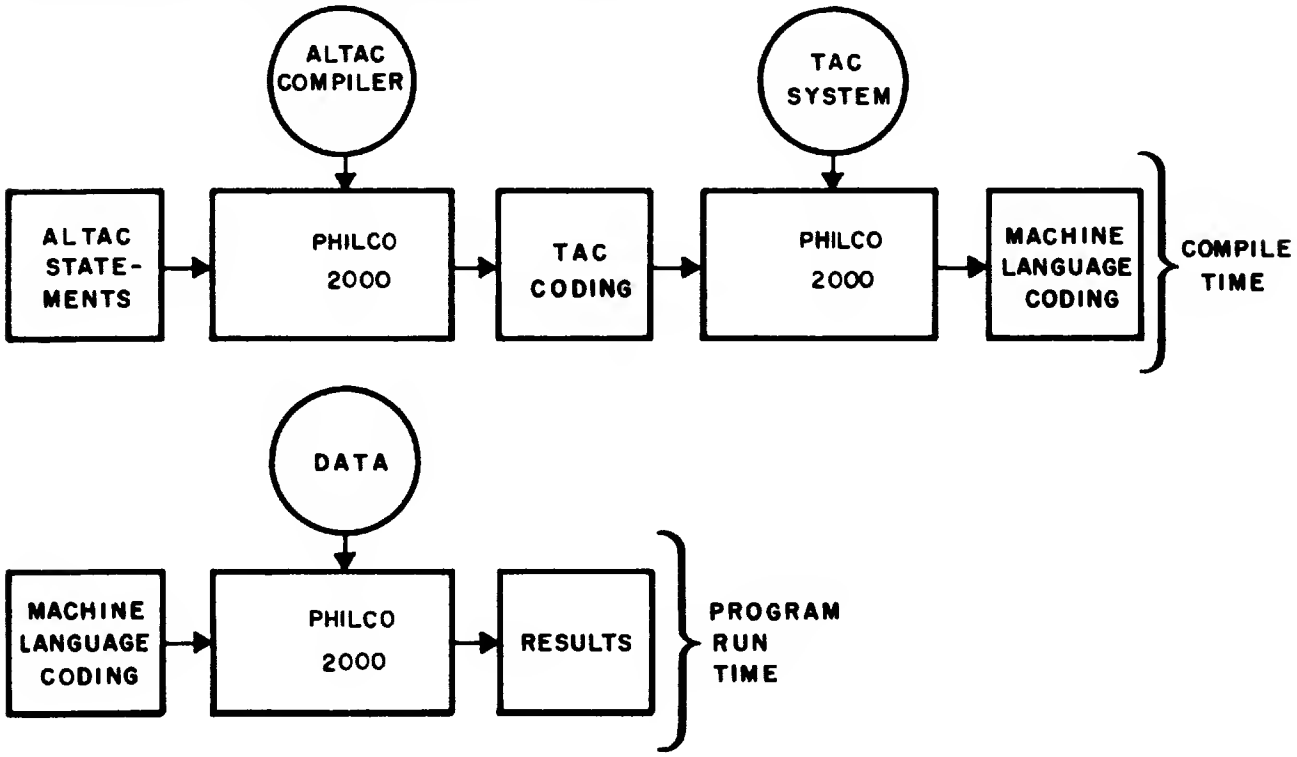
TABLE OF CONTENTS (Cont'd)

TABLE OF CONTENTS (Cont'd)

# INTRODUCTION

This manual contains a detailed description of the rules that must be followed in writing programs within the limits of the ALTAC language.

The ALTAC (ALgebraic translator into TAC) compiler accepts algebraic statements and converts them to TAC language coding. These algebraic statements may be in either ALTAC or existing FORTRAN* card format. The TAC System then operates in the usual manner to produce a running machine language program. The following diagram demonstrates this process, which is of course continuous.




Each Philco 2000 illustrated in the diagram may be a different machine.

The ALTAC System is well adapted to applications of a scientific nature. Operational programs can be prepared in a fraction of the time needed to code directly in TAC code. Furthermore, the TAC coding thus produced will maintain a high degree of efficiency.

---

* Automatic coding system used by International Business Machines Corporation.

Calculations will be performed primarily in floating-point arithmetic with a retention of 10 decimal digits of accuracy. Provision is also included for integer arithmetic.

The ALTAC System provides several important features over other algebraic compilers. These features are covered in detail in the text. The most outstanding of these are:

1) Use of symbolic as well as decimal statement addresses.

2) A sequence of statements may be written in compound form.

3) Conditional statements provide extreme flexibility.

4) Easy incorporation of TAC coding into ALTAC programs.

5) Limitations on form and use of subscripts are minimized.

6) "Mixed" expressions are permitted.

# CHAPTER 1 - SOLUTION OF A PROBLEM USING THE ALTAC SYSTEM

The conventional procedure in solving a given problem on any computer consists of the following general steps:

1) Analysis of the problem and formulation of a solution.

2) Planning the most efficient method of computer solution.

3) Translation of the problem into a language meaningful to the computer.

4) Testing and correction of the program (debugging).

5) Entry of instructions, data into the computer.

6) Output and analysis of results.

At Step 3, the problem must be translated into a language capable of being reduced to machine language. Instructions may be written in TAC mnemonic code for translation by the assembler-compiler into instructions in machine language. The algebraic formula $Y = X^2 - 3X + 5$, for example, would be coded in the TAC language as follows:

| COMMAND | ADDRESS AND REMARKS |
|---|---|
| T MQ | F / 3 $ |
| F MMR | X $ |
| T MQ | X $ |
| F MS U | X $ |
| F A M | F / 5 $ |
| T A M | Y $ |
| | |
| | |
| | |

The same algebraic formula would be written in the ALTAC language as follows:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | Y = X**2 - 3*X + 5$ |
| | |

where ** denotes "raising to the power," and * multiplication.

The statement would then be punched in Hollerith on a card and entered into the computer. The ALTAC compiler converts the statement to TAC language coding. The final step in the process reduces the TAC language coding to machine language instructions capable of being entered into the Philco 2000 and executed.

## ALTAC CODING FORM

An ALTAC program consists of a series of statements written in a logical order. The format of the coding form is as follows:

| IDENTITY AND SEQUENCE | L (9) | LOCATION (10) | ALTAC STATEMENT (17) |
|---|---|---|---|
| | | | |
| | | | |
| | | | |

| COLUMNS | HEADING | CONTENTS AND DESCRIPTION |
|---|---|---|
| 1-8 | IDENTITY and SEQUENCE | Used for labeling purposes. The compiler will ignore any punching in these columns. |
| 9 | L | An asterisk (*) in this column will cause the compiler to ignore any punching on this card, and any punching will be interpreted as a remark. |
| 10-16 | LOCATION | Used for statement references, which may be any five or less unsigned decimal digits or any symbol acceptable to TAC. |

| COLUMNS | HEADING | CONTENTS AND DESCRIPTION |
|---|---|---|
| 17-80 | STATEMENT | All statements are written in these columns. Blank spaces are ignored during the compilation process. |

Semicolons are used to separate multiple statements. Such statements may be continued onto the next and succeeding cards starting in column 17. The first $ character encountered marks the end of the statement or series of statements.

## ALTAC CHARACTERS

Allowable characters in ALTAC are the 10 decimal digits 0-9, 26 alphabetic characters, and the special characters + - * / ( ),. ;= blank and $.

# CHAPTER 2 - BASIC ELEMENTS OF THE ALTAC LANGUAGE

Since arithmetic is performed by the Philco 2000 in two modes, fixed point and floating point, the ALTAC language provides rules for the unique expression of both types of numerical quantities. These may be represented as constants and variables. When linked together with "operators," these constants and variables form expressions meaningful to the ALTAC compiler, as will be seen in Chapter 3.

## CONSTANTS

ALTAC constants can be one of two types – fixed-point constants or floating-point constants. They are expressed in the language of ALTAC in much the same manner as in the language of mathematics. It is the form in which they are written that determines the mode in which they are interpreted.

## FIXED-POINT CONSTANTS

Fixed-point constants are written as positive or negative decimal integers. Since they are stored in the 16 left-most bits of a memory location (sign and 15 bits), the range of integers that can be expressed is -32768 to + 32767.

The following are examples of fixed-point constants:

1, -50, 32767, 389, -20

Plus signs are understood when omitted.

## FLOATING-POINT CONSTANTS

Floating-point constants are written as decimal integers, and are distinguished from fixed-point constants by the appearance of a decimal point in any position in the field. The range of a floating-point number, N, is:

$$-1 \times 2^{2047} \leq N < + 1 \times 2^{2047}$$

A signed or unsigned scaling factor, E, with a decimal exponent may follow the number. Such an exponent must not exceed ± 600 in magnitude.

Examples of floating-point constants are:

1., 3.14159, -.0062

These numbers could also have been written with an E as follows:

10.0 E-1, .314159 E1, -6.2 E-3

which would result in the same floating-point numbers.

## VARIABLES

Both fixed-and floating-point numbers may be represented in a more general manner by writing symbols in place of the actual numbers. Such symbols are called variables, and are actually symbolic addresses of memory locations containing the quantities.

Variables consist of from one to a maximum of seven alphanumeric characters. The first character of a variable must be alphabetic for it determines the mode (fixed or floating point) in which the quantity is represented in a memory location. The remaining characters may be any combination of a alphanumeric characters. As will be pointed out later, symbols may also be used as references to functions and to statements in a program, in which case they have different meanings.

## FIXED-POINT VARIABLES

If the first character of a variable is I, J, K, L, M, or N, ALTAC will treat it as a fixed-point variable.

Examples: J, NUMBER, and MEAN3 represent addresses of fixed-point variables. Numbers in these locations occupy the 16 left-most bits of a memory location as do fixed-point constants. Numbers that fall outside the range indicated for fixed-point constants will be reduced modulo $2^{15}$.

## FLOATING-POINT VARIABLES

When the first character of a variable is any alphabetic character other than I, J, K, L, M, or N, it is treated as a floating-point variable.

Examples: A, BETA, and DØLLAR4 represent addresses of floating-point variables. Floating-point quantities represented by variables must lie within the same range as floating-point constants.

## SUBSCRIPTS

While variables may be used to refer to single quantities, or locations in memory, they may also be made to represent n-dimensional arrays of

information in memory, by attaching subscripts to them. These subscripts may be any expression with fixed-point integer values that is enclosed in parentheses following the variable.

Examples: A(I), BETA(J), DØLLAR(1), ITEM(2*I+3), MATRIX (K, L, M, N)

Thus, A(I) is the ALTAC representation of $A_i$. A fixed-point constant and an arithmetic expression, as well as a variable, may be used as a subscript to a variable. A maximum of four subscripts may be appended to a variable. Subscripts may be attached to subscripts to any desired depth, as in the following example:

MATRIX (J(I), K)

which is read as MATRIX $_{j_i,\ k}$

J represents an array and, as such, must be dimensioned (see DIMENSION statements, p. 36). ALTAC will pick up the contents of the $i^{th}$ element of array J in fixed point and will use it as a subscript of MATRIX.

A general method for arriving at the effective address represented by a subscript or group of subscripts is presented on p. 37.

# CHAPTER 3 - ALTAC EXPRESSIONS, ARITHMETIC FORMULAS, AND FUNCTION DEFINITIONS

In this chapter, it will be seen how meaningful expressions are formed by linking together a sequence of constants and variables with operation symbols. As a result, formulas and functions can be defined as expressions in a form closely resembling the language of mathematics.

## OPERATION SYMBOLS

In ALTAC, there are five basic "operation" symbols (or characters) which are defined as follows:

| SYMBOL | MEANING |
|---|---|
| + | ADD |
| - | SUBTRACT |
| * | MULTIPLY |
| ** | RAISE TO A POWER |
| / | DIVIDE |

## ALTAC EXPRESSIONS

Any of the above operators define relationships between any sequence of constants, subscripted or non-subscripted variables, and functions, and may be used with commas and parentheses to produce meaningful ALTAC expressions.

For example, the simple mathematical expression,

$$X^2 + X^{3/2} + 3X - 3/4$$

when written in the ALTAC language would appear as follows:

```
X**2.+X**(3./2.)+ 3.*X - 3./4.
```

or

```
X**2.+X**1.5 + 3.*X - 3./4.
```

## MIXED EXPRESSIONS

A special feature in the ALTAC System allows the programmer to write expressions in "mixed" modes. A mixed expression is defined as one containing a combination of fixed-and floating-point variables. The result of the arithmetic would be in floating-point form, unless the expression is made equivalent to a fixed-point variable.

For example, if the previous expression were written in the form:

```
X**2.+X**1.5 + 3.*X - L
```

it would be treated by the compiler as a mixed expression, since L is the only fixed-point variable present. In any mixed expression, as in this example, the fixed-point quantities are converted to floating point before the arithmetic is performed. If the constant of the X term were written without the decimal point, a floating-point constant would be formed during the compilation process. In general, when at least one floating-point variable appears in an expression, or within parentheses, the floating-point mode has precedence.

Examples: A floating-point value would result from the following cases:

```
A ** I
A ** B
I ** B
```

while a fixed-point value would result from the case,

```
I ** N
```

## ARITHMETIC STATEMENTS

An arithmetic formula is expressed in the ALTAC language by writing a variable equal to an expression, in the form V = E. This is read as "compute the value of the expression, E, and store the result in the location represented by the variable, V." The result is always converted to the mode of V.

Thus, the formula I= nX + 5 when written as an ALTAC statement would appear as:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | I = N*X + 5 $ |
| | |
| | |

The ALTAC compiler would then examine this statement and generate TAC coding as follows:

| COMMAND | ADDRESS AND REMARKS |
|---|---|
| TMQ | N$ |
| FMMR | F/3276 8$ |
| TAM | 0001FIX$ |
| TMQ | 0001FIX$ |
| FMMR | X$ |
| FAM | F/5$ |
| XFIX | $ |
| TAM | I$ |
| | |
| | |

Other examples of statements involving mixed expressions are:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | Y = I - L $ |
| | L = I + F $ |
| | A(K) = 3*1.025$ |
| | AVERAGE = TØTAL/N$ |
| | |
| | |
| | |

The second example, L = I + F, represents a special situation. Note that L is a fixed-point variable. In such a case, the value of the expression

(I + F) would be computed in floating point; then, the result would be converted to fixed point, truncated to an integer, reduced modulo $2^{15}$, and stored in the 15-bit left address of location L.

## COMPOUND STATEMENTS

Compound statements are formed by linking together a series of statements by semicolons on a single line of the ALTAC coding form. Such statements may be continued on succeeding lines starting in column 17. The last statement of the series must be terminated by a $ character.

An example of a compound statement is:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | X = 3*Y; A = 2**I; C = A + B $ |
| | |
| | |

Each statement is executed in the resulting program in the order in which it occurs.

## ORDER OF OPERATIONS

Parentheses may be used to define the order in which a sequence of operations is to be performed. If parentheses are not used, the normal order would be:

1) Raising to a power

2) Multiplication and division

3) Addition and subtraction

As an example, if an expression is written in the form

$$X + Y + Z/2$$

the operations would be performed in machine language as indicated by the grouping

$$(X+ Y) + Z/2$$

$$\overset{3}{\overbrace{\overset{2}{\overbrace{(X+ Y)}} + Z/2}}$$

If it is desired to compute $\frac{X+Y+Z}{2}$, the statement must be written in the form (X+ Y + Z)/2.

When similar operations are repeated in an arithmetic expression, the expression would be "scanned" from left to right, performing the operations by taking two variables at a time as indicated by the following example:

The expression V1oV2oV3oV4oV5 where "o" means "operator" would result in a machine language program "operation nesting"

(V1oV2) ⟶ R1

(R1oV3) ⟶ R2

(R2oV4) ⟶ R3

(R3oV5) ⟶ FINAL RESULT

Parentheses may be used in an expression provided that for every left parenthesis there is a corresponding right parenthesis. If this condition is not met, errors in compilation will occur.

Expressions involving repeated raising to a power cannot be written in ALTAC as X**Y**Z; they will be meaningless to the compiler in this form. Consequently, such operations must be defined by the use of parentheses, in one of the two following forms:

(X**Y)**Z meaning $(X^{Y})^{Z}$

X**(Y**Z) meaning $X^{(Y^{Z})}$

## FUNCTIONS AND FUNCTION STATEMENTS

Functions are a predetermined sequence of operations defined and executed outside the main body of a program, and may be called upon from the main program to perform a series of calculations as needed. There are three distinct types of functions in ALTAC:

1) Functions defined by a single program statement.

2) Library functions.

3) Functions defined by separate programs.

The name of a function is composed of from one to seven alpha-numeric characters. The first character, which must be alphabetic,

determines the mode of the value of the function. The following rules are to be observed when naming functions:

Rule 1.

If the name of a function is 4 to 7 characters long, with the last character an F, the value of the function is in fixed-point mode if and only if the first character is X.

Rule 2.

If the name of a function is less than 4 characters long or if the last character is not F, the value of the function is in fixed-point mode if and only if the first character is I, J, K, L, M, or N.

Examples of function references:

SQRTF (B**2 - 4* A * C)

VALUE (X, M)

VALUEF (X, M)

IVALUE (X, M)

XVALUEF (X, M)

The values of the first three functions are in floating point while those of the last two functions are in fixed point. The arguments of each function are enclosed in parentheses and are separated by commas.

## FUNCTIONS DEFINED BY A SINGLE PROGRAM STATEMENT

While the first example above is used to refer to the square root function on the library tape, the last four refer to a function defined by a statement in the program. Note that while VALUE or VALUEF will yield floating-point results, the values of IVALUE and XVALUEF will be in fixed-point mode. A statement defining the last function may be written as follows:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | XVALUEF(A, N) = P* A + N**2 $ |
| | |
| | |

Such a function definition statement must precede the first executable statement in the program. In the main body of the program, a statement calling upon the function could be written as:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | RESULT = 2*X**2 + 3*XVALUEF(X, L)$ |
| | |
| | |

The variables X and L, as they appear, are presented to the function XVALUEF as arguments, and have a one-to-one correspondence with the arguments A and N in the function definition statement.

Variables listed as arguments of a defining function (e.g., A and N) are termed "dummy variables." While they need not be the same variables, they must agree in number, order, and mode with the corresponding arguments in the calling statement. Variables, such as P in the example, not listed as arguments are treated as variables, and will assume whatever values they currently have in the program. A statement defining a function may in turn refer to another function of any of the four types.

The arguments of a function reference may take the form of expressions, and may also include subscripted variables as in the following example:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | RESULT= 2*X**2+ 3* XVALUEF(X+4*K, ITEM (L) ) $ |
| | |
| | |

The function XVALUEF would then be computed with the arguments A corresponding to X + 4*K and N corresponding to ITEM (L) — based on

the current values of X, K, $ITEM_1$ and P. The quantities would be used as if the function definition statement had been written in the form:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | XVALUEF(X+4*K, ITEM (L) ) = P * (X+ 4 * K) + |
| | ITEM (L) ) **2 $ |
| | |
| | |

## LIBRARY FUNCTIONS

Library functions are subroutines defined on the TAC library tape. References are made to them in the same manner as for other types of functions. Since these functions will vary among users, each installation will prepare a list of such functions for reference purposes. Standard functions are supplied. The following list describes standard functions and specifies the form in which reference is made.

General Form: FUNCTION NAME (ARGUMENT or ARGUMENTS)

Multiple arguments are separated by commas.

1) FLØATF (ARGUMENT)

   The single fixed-point variable within parentheses will be converted to floating-point form.

2) XFIXF (ARGUMENT)

   The single floating-point variable within parentheses will be converted to fixed point, truncated to an integer value, and reduced modulo 32768.

3) ABSF (ARGUMENT)
   XABSF (ARGUMENT)

   These functions will produce the absolute value of a floating-or fixed-point variable respectively.

4) MAXF (ARGUMENTS)
XMAXF (ARGUMENTS)

The largest value of the arguments, which must be at least two and not over thirty, will be selected. The mode of the arguments and the value of the functions may be either fixed or floating point.

5) MINF (ARGUMENTS)
XMINF (ARGUMENTS)

The smallest value of the arguments will be selected, where there are at least two but not over thirty arguments. The mode of the arguments and the value of the functions may be in either fixed or floating point.

6) MØDF (TWO ARGUMENTS)
XMØDF (TWO ARGUMENTS)

These functions are used to produce integral remainders and can be described as

$$\text{MØDF}\ (\text{Arg}_1, \text{Arg}_2) = \text{Arg}_1 - \text{INTF}\ (\text{Arg}_1/\text{Arg}_2)*\text{Arg}_2$$

$$\text{XMØDF}\ (\text{Arg}_1, \text{Arg}_2) = \text{Arg}_1 - \text{XINTF}\ (\text{Arg}_1/\text{Arg}_2)*\text{Arg}_2$$

7) INTF (ARGUMENT)
XINTF (ARGUMENT)

These functions are used to truncate a floating-point fractional number to the largest integer less than or equal to the absolute value of the argument. The sign of the argument is preserved.

8) SIGNF (TWO ARGUMENTS)
XSIGNF (TWO ARGUMENTS)

These functions are used to transfer the sign of the second argument to the first argument.

9) DIMF (TWO ARGUMENTS)
XDIMF (TWO ARGUMENTS)

These functions are defined as

$$\text{DIMF}\ (\text{Arg}_1, \text{Arg}_2) = \text{Arg}_1 - \text{MINF}\ (\text{Arg}_1, \text{Arg}_2)$$

$$\text{XDIMF}\ (\text{Arg}_1, \text{Arg}_2) = \text{Arg}_1 - \text{XMINF}\ (\text{Arg}_1, \text{Arg}_2)$$

The last type of function, that defined by a separate program, allows function definition by more than one statement. This type is discussed in detail in Chapter 7.

# CHAPTER 4 - ALTAC CONTROL STATEMENTS

This chapter explains the statements the reader will use to control the sequence of flow through a program.

In general, these statements may be used to

1) Set, alter, or test program switches.

2) Test variables and provide conditional jumps.

3) Execute a particular sequence of statements repeatedly.

4) Index subscripted variables.

## UNCONDITIONAL GØ TØ

Form: GØ TØ m

Description:

Control is unconditionally transferred to the statement with address m, where m is either a fixed decimal or a symbolic address.

Examples:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | GØ TØ 9 $ |
| | . . . |
| | . . . |
| 9 | Some statement |
| | GØ TØ ALPHA $ |
| | . . . |
| | . . . |
| ALPHA | Some statement |
| | |
| | |

ASSIGNED GØ TØ

A GØ TØ statement that is subject to modification by an ASSIGN statement is called an ASSIGNED GØ TØ statement.

The ASSIGN statement may be written as,

Form: ASSIGN n to M

where n is a decimal integer, or as

Form: ASSIGN (n) to M

where n is any symbol. M is a dummy variable.

The GØ TØ statement subject to modification by one of these ASSIGN statements may be written in one of two forms:

Form 1: GØ TØ M

or

Form 2: GØ TØ M, $(k_1, k_2, k_3, \ldots k_n)$

where $k_1, k_2, k_3, \ldots k_n$ are statement (decimal) numbers or symbolic addresses representing a list of values which may be assigned to M.

Examples:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | . . . |
| | . . . |
| | Some statement |
| | . . . |
| | ASSIGN 9 TØ M $ |
| | . . . |
| | GØ TØ M |
| | . . . |
| 9 | SUM = 0.0 $ |
| | Some statement |
| | . . . |
| | ASSIGN (ALPHA) TØ M $ |
| REF | Some statement |
| | GØ TØ M, (ALPHA,15,BETA) $ |
| | . . . |
| ALPHA | Some statement |
| | ASSIGN 15 TØ M $ |
| | GØ TØ REF $ |
| 15 | Some statement |
| | ASSIGN (BETA) TØ M $ |
| | GØ TØ REF $ |
| BETA | Some statement |
| | . . . |
| | . . . |

## COMPUTED GØ TØ

Form: GØ TØ ($m_1$, $m_2$, $m_3$, . . . . , $m_n$), I

Description:

This statement functions as a program switch. The possible paths are listed as arguments, i.e., $m_1$, $m_2$, $m_3$, . . . ., $m_n$. The switch is set prior to reaching the GØ TØ statement by assigning a value to I.

Specifically, a jump is made to statements $m_1$, $m_2$, $m_3$, . . . ., $m_n$ at the time of execution of the statement if I has the value 1, 2, 3, . . . ., n. I may be a current index value, a preset value, or a value computed by integer arithmetic. ALTAC allows symbolic addresses to be substituted for $m_1$, $m_2$, $m_3$, . . . ., $m_n$. I should not be zero or a negative quantity. If I is not an integer in the range 1 to n, an error is indicated at run time.

Examples:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | GØ TØ (10, 15, 20), M $ |
| | |
| 10 | Some statement |
| 15 | Some statement |
| 20 | Some statement |
| | |
| | GØ TØ (ALPHA, BETA, GAMMA), L $ |
| | |
| ALPHA | Some statement |
| BETA | Some statement |
| GAMMA | Some statement |
| | |
| | |

## IF STATEMENT

Form 1: IF (e) $m_1$, $m_2$, $m_3$

Description:

If the expression e is negative, a jump is made to $m_1$; if zero, to $m_2$; if positive, to $m_3$.

This statement is used both as a test and a resulting jump at the time of program execution. The expression e may take the form of any variable or arithmetic expression. Symbolic, as well as decimal, statement references may be used for $m_1$, $m_2$, and/or $m_3$.

Examples:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | IF (B (I) - Z) 5, 10, 20 $ |
| 5 | Some statement |
| 10 | Some statement |
| 20 | Some statement |
| | IF (B(I) - Z) ALPHA, BETA, GAMMA $ |
| ALPHA | Some statement |
| BETA | Some statement |
| GAMMA | Some statement |
| | |
| | |

## IF STATEMENT

Form 2: IF ($e_1$) r ($e_2$), t

Description:

The expression $e_1$ is compared with the expression $e_2$, using a relationship r as a test. If the two expressions satisfy the criterion of the relationship, imperative statement t is executed. If the test is not satisfied, a jump is made to the next statement in the program.

Let $e_1$ and $e_2$ be any fixed- or floating-point expressions, and r one of the following relations:

| r | MEANING |
|---|---|
| E | EQUAL |
| NE | NOT EQUAL |
| LT | LESS THAN |
| LTE | LESS THAN OR EQUAL |
| GT | GREATER THAN |
| GTE | GREATER THAN OR EQUAL |

If the relationship between $e_1$ and $e_2$ is satisfied, the imperative statement t is performed. Statement t may be a GØ TØ or an arithmetic statement.

Compound conditional statements (where $e_1$, $e_2$, . . . . are expressions, $r_1$, $r_2$, . . . . relations, and $t_1$, $t_2$, . . . . imperative statements) may be written in the ALTAC language as follows:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | IF (E1) r(E2), T1; IF (E3) r(E4), T2; IF (E5) r(E6), |
| | T3;. . . . . $ |

Thus, a compound conditional statement is one composed of several conditional statements separated by semicolons. The imperative statements $T_1$, $T_2$, . . . . may themselves be compound statements.

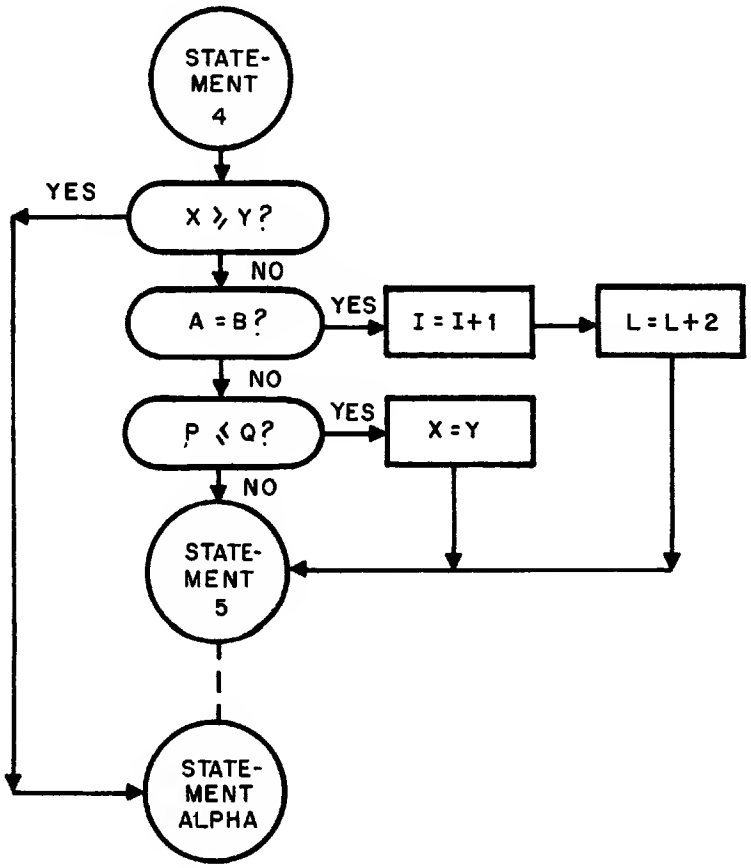
The running program tests the conditional statement; if the condition is not satisfied, the next conditional statement of the compound conditional statement is tested and the procedure is repeated until no more conditional statements remain to be tested. The statement following the compound conditional statement is then executed.

If any of the conditional statements is satisfied, the imperative statement following the satisfied conditional statement is executed, and control is transferred to the statement following the compound conditional statement.

An example of a compound IF statement is:

| LOCATION | ALTAC STATEMENT |
|---|---|
| 4 | IF (X) GTE (Y), GØ TØ ALPHA; IF (A) E (B), I = I + 1; |
| | L = L + 2; IF (P)LTE(Q), X = Y$ |
| 5 | Next Statement |
| | . . . |
| | . . . |
| | . . . |
| | . . . |
| | . . . |
| | . . . |
| ALPHA | Some statement |
| | |

This statement operates in the manner indicated by the following flowchart:



## DØ STATEMENT

Forms: DØ m n = $v_1$, $v_2$, $v_3$

DØ (m) n = $v_1$, $v_2$, $v_3$

Description:

All of the statements following the DØ statement, up to and including statement m are repeated. The number of repeats is determined by the index value of n, where $v_1$ is the initial index value, $v_2$ is the limit of the index value, and $v_3$ is the indexing increment. The last statement, m, of the DØ loop cannot be an IF statement, since the normal proceeding of an IF statement to the next statement which follows in sequence would take it outside the range of the DØ loop. However, a compound IF statement of the Form 2 type is allowed as the last statement in the range of a DØ.

Statements under the control of a DØ statement are repeated in a manner similar to a loop of TAC instructions utilizing index registers. However, unlike TAC, all information needed to set, modify, and test the index is contained in the DØ statement itself.

All statements appearing in sequence following the DØ statement up to and including statement m are said to be under control of the DØ statement, and are repeated until the DØ indexing is satisfied. The statement immediately following statement m is the first statement outside the range of the DØ and is executed when the DØ is satisfied.

In a DØ statement, m may be either a statement number or symbolic address. If m is a symbolic address, it must be enclosed in parentheses. The index, n, is written as a non-subscripted fixed-point variable. The parameters of the index, $v_1$, $v_2$, and $v_3$, are written as fixed-point constants, or variables with pre-assigned integer values. $v_1$, $v_2$, and/or $v_3$ must be positive non-zero integers to achieve normal indexing. The index, n, is initially set to the value $v_1$. Each time statement m at the end of the range is completed, n will be increased by the increment $v_3$. Statements under control of the DØ statement are executed for all values of n less than or equal to the value of $v_2$. For the first value of n greater than $v_2$, control will pass out of the range of the DØ to the next statement in the program. Thus $v_2$ defines the largest value that the index n can attain in the range of a DØ statement. If a DØ statement is written in the form DØ m n = $v_1$, $v_2$ the increment of the index will automatically be taken to be 1.

During the execution of statements within the range of a DØ, the current value of the index, n, may be used as a subscript of a variable or as a

constant in an arithmetic statement. However, any calculation within the range cannot change its value; modification may be made only by the normal DØ indexing operations provided in the DØ statement itself.

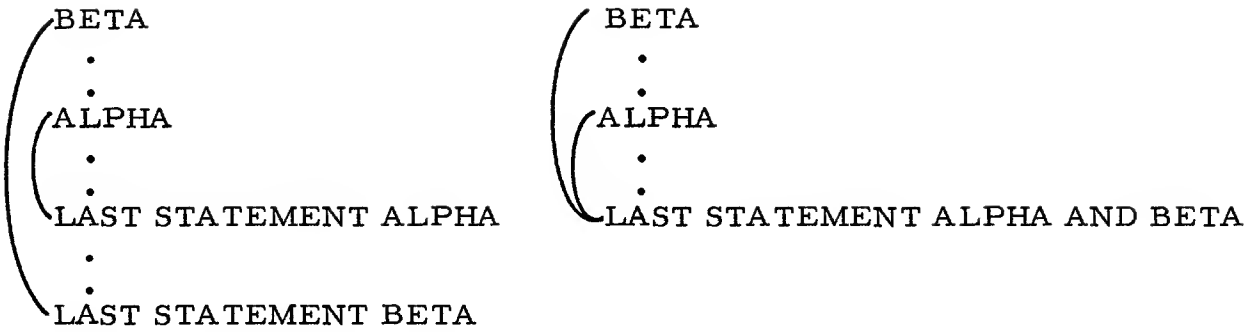
The following is an example of the use of a DØ to compute the sum of a series of numbers:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | N = 200 $ |
| | SUM = A(1) $ |
| 2 | DØ (END) I = 2, N $ |
| END | SUM = SUM + A(I) $ |
| 3 | Next Statement |
| | |
| | |

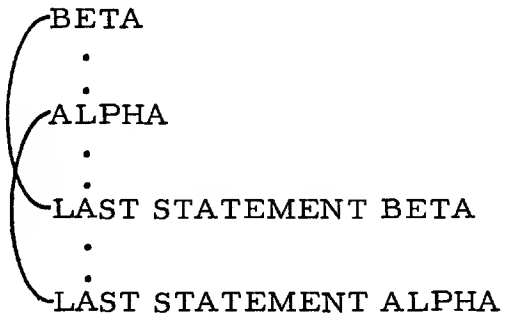
N, the number of elements in the array A to be summed, is assigned a value prior to reaching statement 2. Statement 2 causes the statement with the symbolic address END to be executed repeatedly. I is assigned an initial value of 2, and increased by 1 after each execution of statement END. When I has reached the value N, the statement is executed for the last time, and control is then transferred to statement 3 of the program, out of the range of the DØ.

Calculations involving arrays of more than one dimension may likewise be indexed by writing statements involving their names with multiple subscripts. Such statements are then included within the range of several DØ's. Multiple DØ statements may be written in ALTAC, but limited overlapping of ranges is permitted. This means that when a DØ statement is within the range of another DØ, all statements within the range of the "inner" DØ must also be within the range of the "outer" DØ.

For example, assume that ALPHA and BETA are two DØ statements with different parameters which are to be used to set up a loop. Then the following types of nesting arrangements are permitted.




An example of a sequence which is not allowed is:




For, in the resulting program, the indexing parameters would not be properly modified.

Jumps into the range of a DØ statement from outside its range are not permitted. Thus, in a "nest" of DØ's, jumps can be made from within the range of an "inner" DØ statement to a statement within the range of an "outer" DØ statement, but not vice-versa.

There is one exception to this rule, however. If a jump has been made from within the range of a DØ statement to a section of the program completely outside the range of the DØ statement, and a series of calculations is performed in this section which does not modify any of the indexing parameters of the DØ, then control may be properly transferred back into the range of the same DØ statement from which exit was made.

If the DØ indexing has been satisfied and control has been transferred out of the range of the DØ statement, the value of the index controlled by the DØ is not defined, and must be redefined before it is used again. But, if control is transferred out of the range before the DØ indexing is satisfied, the current value of the index is available for use.

## SENSE LIGHT

A sense light is represented in the Philco 2000 as one bit of a word. This bit may be one or zero simulating an on or off condition. The lights or bits are numbered 1 to 48 from left to right, and are used as program switches.

Form: SENSE LIGHT n

Description:

This statement is used to turn sense lights on and off, i.e., set bits to one or zero. If n has the value zero, all lights are turned off. Any other integer value of n from 1 to 48 causes that particular bit to be set to 1.

Examples:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | SENSE LIGHT 0 $ |
| | SENSE LIGHT 45 $ |
| | |
| | |

## IF SENSE LIGHT

This statement provides a means of testing the condition of a sense light.

Form: IF (SENSE LIGHT n) $m_1$, $m_2$

Description:

Sense Light n is interrogated, where n may be any integer from 1 to 48. If the corresponding bit is set, a jump is made to statement $m_1$, and the bit is unset; if the bit is not set, a jump is made to statement $m_2$. Symbolic addresses may be used for $m_1$ and $m_2$.

Examples:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | IF (SENSE LIGHT 2) 5, 10 $ |
| | . . . |
| 5 | Some Statement |
| | . . . |
| 10 | Some Statement |
| | . . . |
| | IF (SENSE LIGHT 47) ALPHA, BETA $ |
| | . . . |
| ALPHA | Some Statement |
| | . . . |
| BETA | Some Statement |
| | |
| | |

## IF SENSE SWITCH

A Sense Switch is one of the 48 toggle switches on the Philco 2000 console.

Form: IF (SENSE SWITCH n) $m_1$, $m_2$

Description:

This statement interrogates one of the 48 toggle switches. The switches are numbered 0 to 47 from left to right. If the specified toggle switch is on, jump is made to statement $m_1$; if off, to statement $m_2$. The reference n may be any integer or fixed-point variable (assigned or computed), whose value is any integer 0-48, with 48 being interpreted as switch 0.

Examples:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | IF (SENSE SWITCH 1) 5, 10 $ |
| | . . . |
| 5 | Some Statement |
| | . . . |
| 10 | Some Statement |
| | . . . |
| | IF (SENSE SWITCH 47) ALPHA, BETA $ |
| | . . . |
| ALPHA | Some Statement |
| | . . . |
| BETA | Some Statement |
| | |
| | |

## IF SENSE BIT

The IF SENSE BIT statement is used to test the contents of particular bits of decimal location 49 (Octal 61) in memory. This statement is written as follows:

Form: IF (SENSE BIT n) $m_1$, $m_2$

Description:

This statement causes transfer of control to statements $m_1$ or $m_2$, if bit n of memory location 49 is 1 or 0, respectively. (Bits 0 and 48 refer to the sign bit.)

The reference n may be any integer 0-47, or a fixed-point variable. $m_1$ and $m_2$ may be statement numbers or symbolic addresses.

Examples:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | IF (SENSE BIT 3) 5, 10 $ |
| | . . . |
| 5 | Some Statement |
| | . . . |
| 10 | Some Statement |
| | . . . |
| | IF (SENSE BIT 40) ALPHA, BETA $ |
| | . . . |
| ALPHA | Some Statement |
| | . . . |
| BETA | Some Statement |

## IF ØVERFLØW

This statement tests for floating-point exponent overflow.

Form: IF ØVERFLØW $m_1$, $m_2$

Description:

If floating-point exponent overflow has occurred, and the overflow indicator has been set, a jump is made to statement $m_1$ and the overflow indicator is cleared; if not, a jump is made to statement $m_2$.

Examples:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | IF ØVERFLØW 5, 10 $ |
| | . . . |
| 5 | Some Statement |
| | . . . |
| 10 | Some Statement |
| | . . . |
| | IF ØVERFLØW ALPHA, BETA $ |
| | . . . |
| ALPHA | Some Statement |
| | . . . |
| BETA | Some Statement |
| | |
| | |

CØNTINUE

Form: CØNTINUE

Description:

This statement is one which does not produce instructions in the program. Its principal use is as the last statement within the range of a DØ. It provides a common location to which transfer may be made from several statements within the range of the DØ, so as to modify and test the index.

Suppose, for example, that a conditional transfer statement is contained within the range of a DØ, and that one of three separate sequences of operations is to be performed depending upon the outcome of the test. The last statement of each block of operations could not be used for indexing and testing purposes, since m (in DØ m) can refer only to one fixed statement and cannot itself be modified. Therefore, it is necessary to provide a common statement to which a jump may be made from at least two of the three independent groups of operations so as to modify and test the index without performing any operation. In this case, the statement to which m

refers would be a CØNTINUE statement — interpreted as: "do nothing, but proceed to modify and test the index."

PAUSE

This statement is used to provide a temporary halt in a program. No runout is produced. (The reader is referred to the IØPS Manual for a discussion on RUNØUT.)

Form: PAUSE m

Description:

When this statement is reached, the program will halt. The octal number m will be displayed in the program register. (m may be up to 5 octal digits; if omitted, zero is assumed.) Pressing the ADVANCE bar on the console will cause the program to resume, starting at the next statement.

Example:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | PAUSE 777 $ |
| | |
| | |

STØP

This statement is used to provide a final halt in a program. The statement produces automatic runout of all output units described in the IØUNITS statement (see p. 59).

Form: STØP m

Description:

When this statement is reached, the machine will halt, and the number m, which must be written in octal, will be displayed in the program register. (m may be up to 5 octal digits; if omitted, zero will be assumed.) At this point, it will be impossible to continue.

Example:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | STØP 333 $ |
| | |
| | |
| | |

# CHAPTER 5 — ALTAC SPECIFICATION STATEMENTS

The ALTAC system includes four specification statements:

1) EQUIVALENCE Statements

2) CØMMØN Statements

3) DIMENSIØN Statements

4) TABLEDEF Statements

These statements are called specification statements because in allocating storage, they specify the arrangement of data in memory. In a program an EQUIVALENCE statement must precede a CØMMØN, DIMENSIØN or TABLEDEF statement. A CØMMØN statement, in turn, must precede a DIMENSIØN or TABLEDEF statement.

## DIMENSIØN STATEMENTS

In the compilation process, ALTAC will assign a separate memory location for each constant and non-subscripted variable appearing in the program. Groups of memory locations will also be assigned for arrays, the elements of which are referred to by subscripted variables. In the latter case, information specifying the maximum size of each array must be furnished. This is accomplished by writing a DIMENSIØN statement containing a list of the variable names (followed by the maximum values of the subscripts) written as fixed decimal constants and presented as arguments within parentheses. The form of such a statement is as follows:

DIMENSIØN variable (size), variable (size),.....$

Alternatively, a separate DIMENSIØN statement may be written for each array. The limit of four dimensions applies to DIMENSIØN statement arguments as well as to variable subscripts. A DIMENSIØN statement for an array must appear before any statement in the program which refers to one of the elements of that array. Also, subscripted references in a program must agree in number of dimensions or subscripts.

For example, consider BETA, GAMMA, and DELTA as three groups of size (5 x 7), 5, and 5 respectively. Their DIMENSIØN statement would appear as follows in a theoretical program:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | DIMENSIØN BETA (5, 7), GAMMA (5), DELTA (5) $ |
| | . . . |
| | . . . |
| | DELTA (I) = BETA (I, J) + GAMMA (I) $ |
| | |
| | |
| | |

It is sometimes necessary for a programmer to compute the address associated with a subscripted variable. To this end, the following information should be of assistance.

General equation for computing the address of a subscripted variable:

General Form of DIMENSIØN Statement: DIMENSIØN $A(N_1, N_2, N_3, N_4)$

General Form of Reference: $A(I_1, I_2, I_3, I_4)$

General Equation for Computing Effective Address:

$$A(I_1, I_2, I_3, I_4) = A+(I_1-1)+N_1(I_2-1)+N_1N_2(I_3-1)+N_1N_2N_3(I_4-1)$$

For a variable of less than four dimensions, assume the unused subscripts as equal to 1 when used in the general equation.

Example:

The equation for the subscripted variable, A(i, j, k), dimensioned as $A(N_1, N_2, N_3)$ is

$$A(i,j,k) = A+(i-1)+N_1(j-1)+N_1N_2(k-1)$$

If the dimensions were expressed numerically as DIMENSION A(2, 2, 2) the address of A(2, 1, 2) = A+(2-1)+2(1-1)+4(2-1) = A+5, the sixth element of array A.

## EQUIVALENCE STATEMENTS

Normally, the ALTAC compiler will reserve a separate memory location for each distinct variable and dimensioned block which has been defined in the program.

However, a situation may exist, especially in the case of a large program, where it would be necessary and/or convenient to be able to multiple-reference certain memory locations. If the logic of the program permits, this can be accomplished by the use of an EQUIVALENCE statement, which in effect, conserves memory space.

The form of such a statement presents, as arguments, quantities which would then be assigned the same memory locations. The statement must appear at the beginning of the program before any subscripts are used, and must be of the following general form:

EQUIVALENCE (V1, V2, V3, ... ), (V4, V5, V6, ...)$

V1, V2, V3 .... V4, V5, V6 may be single variable quantities or members of groups of locations. They may not be constants.

Example:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | EQUIVALENCE (ALPHA, BETA (6), GAMMA (3) ) $ |
| | |
| | |
| | |

Assuming that ALPHA is the location of a single non-subscripted variable and that BETA and GAMMA represent arrays dimensioned 7 and 5 respectively, locations would be assigned as follows:

| RELATIVE LOCATION | STORAGE ASSIGNMENT |
|---|---|
| (P) | BETA |
| (P) + 1 | BETA + 1 |
| (P) + 2 | BETA + 2 |
| (P) + 3 | BETA + 3, GAMMA |
| (P) + 4 | BETA + 4, GAMMA + 1 |
| (P) + 5 | ALPHA, BETA + 5, GAMMA + 2 |
| (P) + 6 | BETA + 6, GAMMA + 3 |
| (P) + 7 | GAMMA + 4 |

Since locations (P) + 3, (P)+ 4, (P) + 5, and (P) + 6 may be multiple-referenced, the programmer must insure that the proper values appear in these locations at the time of reference.

In summary, all subscripted variables used in a program must have appeared, with their maximum values as arguments, in a DIMENSIØN statement.

## CØMMØN STATEMENTS

The CØMMØN statement permits intercommunication between programs and subroutines. It is used to reserve areas of common storage (outside the boundaries of the object program) that are equally accessible both to main programs and to subprograms.

The CØMMØN statement is written as follows:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | . . . |
| | CØMMØN A,B,C $ |
| | . . . |

The parameters A, B, and C are names of single non-subscripted variables or names of dimensioned arrays. The variables appear in common storage in the same order that they are placed in the CØMMØN statement, provided EQUIVALENCE is not specified.

CØMMØN areas start at location $1000_{(8)}$ in the 8K version of ALTAC, and at location $6000_{(8)}$ in the 16K or 32K versions of ALTAC.

Variables appearing in a CØMMØN statement that also appear in an EQUIVALENCE statement, will be placed (before the other arrays in the CØMMØN statement) in the common area in the same order in which they appear in the EQUIVALENCE statement.

For example, the statements

| LOCATION | ALTAC STATEMENT |
|---|---|
| | EQUIVALENCE (D,H), (A,F) $ |
| | CØMMØN A,B,C,D,E,$ |
| | DIMENSIØN B(3), C(2), E(2) $ |

would provide the following storage assignments:

| RELATIVE LOCATION | STORAGE ASSIGNMENT |
|---|---|
| (P) | D and H |
| (P)+1 | A and F |
| (P)+2 | B |
| (P)+3 | B + 1 |
| (P)+4 | B + 2 |
| (P)+5 | C |
| (P)+6 | C + 1 |
| (P)+7 | E |
| (P)+8 | E + 1 |

The size of EQUIVALENCE plus the total size of those arrays which appear in CØMMØN and do not appear in EQUIVALENCE is the size of the area of memory reserved for common storage.

With RPL and ABS programs, if the size of CØMMØN (determined by the method in the preceding paragraph) is smaller than that desired by the programmer, he can include an additional parameter, nW, in the IDENTIFY statement (IDENTIFY mK, nX, nW $) to insure that the compiled program will have at least n words of common storage. (See p. 57 for a discussion of the IDENTIFY statement.)

## TABLEDEF STATEMENTS

If a programmer wishes to define an array by means of TAC statements, the array must be defined in a TABLEDEF statement. ALTAC does not reserve storage for any array defined in the TABLEDEF statement, unless the array also appears in EQUIVALENCE or CØMMØN statements.

The general form of the TABLEDEF statement is:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | . . . |
| | TABLEDEF Variable (Size), Variable (Size), ....$ |
| | . . . |

Examples:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | . . . |
| | . . . |
| | DIMENSIØN A(10), BETA(5,8), DELTA(4,3,5)$ |
| | TABLEDEF GAMMA(50), KAPPA(7,10)$ |
| | . . . |
| | . . . |

According to the above example, 10, 40 and 60 locations would be reserved for arrays A, BETA and DELTA respectively; while for arrays GAMMA and KAPPA, no storage locations would be reserved.

Care must be taken to see that the TAC insert used to define an array, appears between the statements STARTTAC and ENDTAC or if not, that the TAC insert has a T in the label field (see p. 68).

# CHAPTER 6 — ALTAC INPUT-OUTPUT STATEMENTS

## INTRODUCTION

In this chapter, ALTAC statements which are used to transfer information between core storage in the Philco 2000 and magnetic tape units are discussed.

Information on magnetic tape may be transferred to or from the following external media:

1) Magnetic tape

2) Punched card

3) Printer

The High-Speed Printer is also included as an output device. A complete input or output operation can be directed by writing three special types of statements. They are ORDER, FØRMAT, and ENVIRONMENTAL statements.

An ORDER statement is used to select the appropriate input or output device and to itemize the fields to be transferred. Field and data conversion specifications are provided in a FØRMAT statement. Through the use of DESCRIPTORS and MODIFIERS, information can be transferred between internal (binary) form and external form, with a variety of external formats. Various kinds of ENVIRONMENTAL statements give the programmer operational control of the running program and specify necessary parameters, such as the size of records to be advanced by the ORDER statements.

The programmer has the option of coding input-output operations either directly in ALTAC form or in TAC form. (The reader desiring more information on the inclusion of such TAC statements is referred to the IØPS Manual.)

## ORDER STATEMENTS

An ORDER statement generally contains three parts and performs the following functions:

1) Specifies the appropriate input or output device.

2) Refers to a FØRMAT statement which furnishes field definition and conversion information for certain orders.

3) Provides a list of the quantities to be transmitted.

Because they provide the above features, ORDER statements may be used specifically to:

1) transfer coded information

2) transfer binary information

3) control magnetic tapes

The general form of an order statement used to transfer coded information is:

| LOCATION | STATEMENT |
|---|---|
| NAME | ORDER, FØRMAT NAME, LIST $ |

Commas are used as indicated to separate the statement parts. NAME is the statement number or symbol which may be used by another statement to refer to the ORDER statement. ORDER is a command which specifies the input or output operation and initiates the process.

FØRMAT NAME is a statement number or symbolic reference to a FØRMAT statement to be used in conjunction with the ORDER statement. The statement is completed by a LIST of the symbolic locations which contain or will contain the information to be transmitted. The elements of a LIST are separated by commas.

ORDERS WHICH PROVIDE FOR THE TRANSFER OF CODED INFORMATION

| ORDER | INTERPRETATION | EXAMPLE |
|---|---|---|
| READ | Read Punched Cards | READ, 5, LIST $ |
| PUNCH | Write Punched Cards | PUNCH, 6, LIST $ |
| READ INPUT TAPE n | Read Magnetic Data Tape number n | READ INPUT TAPE K, 11, LIST $ |

## ORDERS WHICH PROVIDE FOR THE TRANSFER OF CODED INFORMATION (Continued)

| ORDER | INTERPRETATION | EXAMPLE |
|---|---|---|
| WRITE ØUTPUT TAPE n | Write Magnetic Data Tape number n | WRITE ØUTPUT TAPE 8, 12, LIST $ |
| PRINT | Print | PRINT, 13, LIST $ |

(n, as shown above and in subsequent orders, may be any integer or fixed-point variable for which a value has been determined.)

When information which will not be converted (binary information) is to be transmitted, the FØRMAT NAME reference and its associated FØRMAT statement must be omitted.

The general form of an ORDER statement specifying the transfer of binary information is as follows:

| LOCATION | STATEMENT |
|---|---|
| NAME | ORDER, LIST $ |

## ORDERS WHICH PROVIDE FOR THE TRANSFER OF BINARY INFORMATION

| ORDER | INTERPRETATION | EXAMPLE |
|---|---|---|
| READ TAPE n | Read Magnetic Binary Tape number n | READ TAPE 3, LIST $ |
| WRITE TAPE n | Write Magnetic Binary Tape number n | WRITE TAPE K, LIST $ |

The general form of an ORDER statement providing control of magnetic tapes is as follows:

| LOCATION | STATEMENT |
|---|---|
| NAME | ORDER $ |

ORDERS WHICH PROVIDE CONTROL OF MAGNETIC TAPES

| ORDER | INTERPRETATION | EXAMPLE |
|---|---|---|
| BACKSPACE n | Backspace Magnetic Tape number n one record | BACKSPACE 5 $ |
| END FILE n | Write End of File Indicator on magnetic tape number n | END FILE 6 $ |
| REWIND n | Rewind magnetic tape number n | REWIND K $ |

## LIST

A LIST included as part of an ORDER statement contains a sequence of subscripted or non-subscripted variables, separated by commas. The order in which information is transferred can be controlled by including indexing information and parentheses in the LIST.

Example:

Three fields of information are punched on a card. It is desired to read this information and store it consecutively in symbolic locations ALPHA, BETA, and GAMMA in the same sequence in which it is punched. The form of the information and field definition on the card will be governed by some FØRMAT statement, which at this point will be referred to as "X". Later, it will be shown how such a FØRMAT statement is written.

The ORDER statement for this operation is written as follows:

STATEMENT

READ, X, ALPHA, BETA, GAMMA $

## INDEXING A LIST

The programmer may specify in a LIST the order in which information is to be transferred. This is done by enclosing subscripted variables in parentheses, along with indexing parameters. Elements of the LIST so defined will be repeated as if they were under the control of a DØ statement. Corresponding pairs of left and right parentheses are used to define the range of each DØ operation.

Example:

Assume that ALPHA and GAMMA represent single memory locations, as in the last example. BETA, however, represents an array of 100 memory locations which have been reserved by a DIMENSIØN statement. The order in which information that is to be read in and stored is punched on cards, is as follows:

ALPHA, BETA(1), BETA(2), ..... BETA(100), GAMMA

The ALTAC ORDER statement is written as follows:

STATEMENT

```
      READ, X, ALPHA, (BETA(I), I = 1, 100), GAMMA $
```

Card after card would be read, and information transferred, until the LIST is satisfied. FØRMAT statement X would contain card field and data conversion specifications. The transfer of elements in the LIST would occur as if they had been written in the following form:

```
      ALPHA
      DØ 4 I = 1, 100
4     BETA (I)
      GAMMA
```

Example:

Assume ALPHA, BETA, and GAMMA are three arrays whose sizes are determined by the following DIMENSIØN statement:

STATEMENT

```
      DIMENSIØN ALPHA(10), BETA(5, 20), GAMMA(20) $
```

The following sequence denotes the order in which the information to be read in is punched on cards:

ALPHA(1), ALPHA(2), ..... , ALPHA (10),

BETA(1, 1), BETA(2, 1), ..... , BETA(5, 1), GAMMA(1),

BETA(1, 2), BETA(2, 2),....., BETA(5, 2), GAMMA(2),

. . . .

. . . .

BETA(1, 20), BETA(2,20),....., BETA(5,20), GAMMA(20)

The ALTAC ORDER statement to read this information from punched cards in the prescribed sequence is written as follows:

STATEMENT

READ, X, (ALPHA(I), I = 1, 10), ( (BETA(I,J), I = 1, 5),

GAMMA (J), J = 1,20) $

Indexing would occur as indicated in the following steps:

| | |
|---|---|
| | DØ 1 I = 1, 10 |
| 1 | ALPHA (I) |
| | DØ 3 J = 1, 20 |
| | DØ 2 I = 1, 5 |
| 2 | BETA (I, J) |
| 3 | GAMMA (J) |

Each pair of parentheses, excluding those containing subscripts, defines the range of a DØ under control of an index. The maximum index values may be variable, as long as the values do not exceed the maximum dimensions of the arrays. Such values may be inserted into the LIST, provided that they appear in sequence before they are used as parameters of the indexing operation.

Example:

STATEMENT

READ, X, L, M, N, (ALPHA(I), I = 1, L), ( (BETA(I, J),

I = 1, M), GAMMA(J), J = 1, N) $

L, M, and N must not exceed 10, 5, and 20 respectively in this case. The maximum sizes of the arrays are determined by a DIMENSIØN statement at the beginning of the program. The first three fields read in are L, M, and N.

## LISTS REPRESENTING ARRAYS

Entire arrays may be transferred by writing a LIST containing references to the names of the arrays as follows:

STATEMENT

READ, X, DELTA, EPSILØN, ØMEGA $

The sequence of reading will then be controlled by the maximum values of the arrays, as specified in their DIMENSIØN statements. The arrays will be filled sequentially in the order in which they appear in the LIST, and their elements stored in order of increasing absolute location.

## FØRMAT STATEMENTS

An ORDER statement involving a transfer of coded information must contain a reference to a FØRMAT statement. FØRMAT statements furnish appropriate field and data conversion information which are needed whenever information that is to be transferred between an external medium and core storage is to be converted from one data form to another; e.g., from coded form to binary. Each FØRMAT statement written defines a record or a number of records to be transmitted. A record is considered to be a punched card of input or output (up to 80 columns), or a printed line of output (up to 120 characters not including printer control characters).

The general form of a FØRMAT statement is as follows:

| LOCATION | STATEMENT |
|---|---|
| NAME | FØRMAT (field descriptors and modifiers) $ |

As in any other type of statement, NAME is the decimal or symbolic address which is used to identify the FØRMAT statement and must appear only in the input or output ORDER statement. The word "FØRMAT" is always written, indicating the type of statement.

FIELD DESCRIPTORS and MODIFIERS are presented as arguments (within parentheses) of FØRMAT statements, and they furnish field specification information. MODIFIERS are written along with DESCRIPTORS, and provide additional flexibility in the input or output operation. Numerous options are provided for their use.

## FIELD DESCRIPTORS

FIELD DESCRIPTORS are used to provide conversion between external form (such as alphanumeric) and internal (binary) form. They are separated by commas in the FØRMAT statement. For clarity, input and output DESCRIPTORS are described separately. The following is a list of DESCRIPTORS and their functions to be covered in this section:

| DESCRIPTOR | INTERNAL FORM | EXTERNAL FORM |
|---|---|---|
| Iw | Integer Binary | Integer Decimal |
| Fw.d | Floating Binary | Fixed Decimal |
| Ew.d | Floating Binary | Floating Decimal |
| nH | Hollerith | Hollerith |
| Ww | Coded Characters | Coded Characters |
| Aw | Coded Characters | Coded Characters |
| nX | Space Symbols | Space Symbols |

### Iw

INPUT: Integer Decimal to Fixed Binary Integer

When used in an input FØRMAT statement, Iw causes a field w columns wide to be converted to a fixed-point binary integer, reduced modulo $2^{15}$ (or 32,768), and stored in the 16 left-most bits (15 bits + sign bit) of a memory location. The w count includes spaces signs and any number of decimal digits that may be within the field. If the field width does not include the sign, the number will be read in as positive. Leading spaces (those before the first non-space character) are treated as spaces, all other space characters are regarded as zeros. The sign, when punched should immediately precede the leading digit.

Examples:

| FIELD | Iw | INTERNAL NUMBER |
|---|---|---|
| 12345 | I 5 | 12345 |
| Δ-12345 | I 7 | -12345 |
| +256ΔΔ | I 6 | +25600 |
| 256 | I 3 | 256 |

Iw

OUTPUT: Fixed Binary Integer to Integer Decimal

Iw causes the quantity occupying the 16 left-most bits of a memory location to be converted to a fixed-point decimal number for output. w specifies the field width in the external medium, which may include spaces and signs. The number produced is positioned with the least significant digit at the right end of the field. If a field specified by w is too small to contain all of the digits to be output, the most significant digits will be lost.

Fw.d

INPUT: Fixed Decimal to Floating Binary

Fw.d converts a fixed decimal data field, w columns in width, to an internal floating-point binary quantity. The right-most d digits in the field are taken as the fractional portion of the number. A decimal point appearing in the w field will take precedence over any d specification. In such a case, the descriptor may be written in the form Fw. (d may be any integer from 1 to 11; if an integer greater than 11 is specified for d, only the first eleven significant digits will be used.)

| FIELD | Fw.d | DECIMAL EQUIVALENT OF INTERNAL NUMBER |
|---|---|---|
| 9.16 | F4 | .916 x 10 |
| -9.16 | F5 | -.916 x 10 |
| - 916 | F4.2 | -.916 x 10 |

The last two fields, as defined by their descriptors, would be converted to identical floating-point numbers.

Fw.d

OUTPUT: Floating Binary to Fixed Decimal

When used as an output descriptor, Fw.d will produce a fixed decimal data field from a floating-point binary quantity occupying a memory location. The resultant field, including sign and decimal point, will be w columns in width, including d decimal digits to the right of the decimal point. The output field will be accurate to ten decimal places. Where more than ten significant digits are requested, the first ten digits will be accurate, and all succeeding digits will be zeros.

## Ew.d

INPUT: Floating Decimal to Floating Binary

Ew.d defines a decimal field in floating-point form, w columns wide, which will be converted to a floating-point number. If an actual decimal point does not appear in the field, the least significant d decimal digits of the mantissa are considered to be the fractional part of the field. If a decimal point appears in the field, it will take precedence over any d specification in the DESCRIPTOR and the .d need not be included. In such a case, the DESCRIPTOR may be written in the form Ew.

The mantissa may be signed or unsigned (if positive). The exponent field follows the last character of the mantissa, and may be separated from it by spaces, all of which are counted in the field width w. The field must be punched in one of the following forms where xxx denotes the numerical exponent:

1) ± mantissa ± xxx

2) ± mantissa Exxx

3) ± mantissa E±xxx

If E is omitted, a sign character must precede the exponent field. The mantissa which should not exceed 11 characters (excluding the sign and decimal point) is converted to a 35-bit rounded quantity. The exponent cannot exceed± 600 in magnitude. The w count includes signs, spaces, mantissa, and exponent. The following are examples of floating decimal input fields with their corresponding DESCRIPTORS:

| FIELD | Ew.d | DECIMAL EQUIVALENT OF INTERNAL NUMBER |
|---|---|---|
| -1.62E+15 | E9 | $-.162 \times 10^{16}$ |
| 15E125 | E6.0 | $+.15 \times 10^{127}$ |
| 234Δ-100 | E8.2 | $.234 \times 10^{-98}$ |
| 2.34Δ+100 | E9.1 | $.234 \times 10^{101}$ |

## Ew.d

OUTPUT: Floating Binary to Floating Decimal

Ew.d defines a decimal field in floating-point form, w columns wide, which will be produced from an internal floating-point number. d specifies the number of decimal digits that will appear to the right of the decimal point. This fractional field will be accurate to 10 decimal places. When a value greater than 10 is specified for d, the first ten digits will be accurate, and all succeeding digits in the fractional field will be zeros. The field to be produced will be in the following form, where xxx denotes the numerical exponent:

±mantissa± xxx

The exponent always contains three numerical digits preceded by a sign, and follows the last character of the mantissa.

The mantissa is of the following general form:

± 0.xxxxxxxxxx

The field width w includes sign, spaces, decimal digits and the decimal point. d specifies the number of decimal digits to the right of the decimal point.

## nH

OUTPUT: Hollerith Characters

nH is a descriptor which is used only with an output statement to define a fixed Hollerith field of n characters (alphanumeric, special and space). The characters are written in the FØRMAT statement immediately to the right of the descriptor. They are thus fixed at the time the FØRMAT statement is written, and are not modifiable at program execution time. Any symbols, including $, may be used with nH.

Example:

A print ORDER statement with the associated FØRMAT statement

FØRMAT (17H ΔΔ PRINTINGΔ SAMPLE) $

would cause the following line to be printed:

PRINTING SAMPLE

## Ww

INPUT or OUTPUT: Coded Characters

Ww is used with both input and output statements. Each Ww descriptor is associated with a word in memory.

Since each Philco 2000 word may accommodate no more than eight binary-coded characters, w should not be greater than 8. Where more than eight (e.g., W12) characters are specified, only the last eight characters will be stored. When less than eight characters are specified, the characters are right-most positioned in the word, and the rest of the word is filled with zeros.

| FIELD | Ww | HOW STORED | EXPRESSION IN MEMORY | Ww | EXTERNAL FORM |
|---|---|---|---|---|---|
| SAMPLE | W6 | OOSAMPLE | SAMPLERS | W6 | MPLERS |
| DATA | W4 | OOOODATA | DATAFORM | W4 | FORM |
| CLASSWORK | W9 | LASSWORK | REPORT A | W9 | Illegal |

If the output of more than eight characters is specified by a single Ww descriptor, an IØPS compilation error will occur.

## Aw

INPUT OR OUTPUT: Coded Characters

Aw is similar in all respects to the Ww descriptor, except that when more than eight characters are specified, only the first eight characters will be stored; and when less than eight characters are specified, the characters are left-most positioned in the word and the rest of the word is filled with space characters.

| FIELD | Aw | HOW STORED | EXPRESSION IN MEMORY | Aw | EXTERNAL FORM |
|---|---|---|---|---|---|
| SAMPLE | A6 | SAMPLE$\Delta\Delta$ | SAMPLERS | A6 | SAMPLE |
| DATA | A4 | DATA$\Delta\Delta\Delta\Delta$ | DATAFORM | A4 | DATA |
| CLASSWORK | A9 | CLASSWOR | REPORT A | A9 | Illegal |

## nX

INPUT or OUTPUT: Space Characters

Unlike the preceding descriptors, nX does not involve conversion of information. This descriptor serves only to space or bypass unwanted characters.

The nX descriptor can be used with input or output statements. When associated with an input statement, actual characters, as they appear in a field, are bypassed and not read into memory. When associated with an output statement, this descriptor causes a field of n spaces to be printed or punched.

## MIXED FIELDS

Mixed fields containing both nH and other descriptors may be specified for output as indicated by the following example:

FØRMAT (9H10ΔΔSUMΔ=, F5.2, 10HΔPRØDUCTΔ= F10.3)$

where Δ indicates a space. This would produce a line of printing in the following format:

SUM =-5.25 PRØDUCT =-25250.325

The descriptors F5.2 and F10.3 correspond to variables specified in the LIST portion of an ORDER statement. ΔΔSUMΔ= and ΔPRØDUCTΔ= are generated under control of their 9H and 10H descriptors, and are not accessible to the program. nH specifies that the following n characters (blanks included) are to be interpreted literally as code mode characters. A comma may be used to separate the end of the nH specification from the following descriptor. Thus, when writing such a FØRMAT statement, an exact count must be made verifying that the (n + 1) significant character following nH is the beginning of the next field description.

## MODIFIERS

FIELD DESCRIPTORS may be further defined by prefixing them with MODIFIERS, which provide added flexibility in controlling input or output operations. Some functions of MODIFIERS are:

1) Repetition of similar field specifications

2) Scaling of numbers in the external medium

3) Editing an output format

Where necessary, several modifiers can be used with a single descriptor. MODIFIERS always precede the DESCRIPTORS they modify, but the order of their appearance is immaterial.

## REPEAT MODIFIER (nR)

A single DESCRIPTOR may be used to define consecutive fields of the same format by prefixing the descriptor with a decimal number, n, which indicates the number of times the DESCRIPTOR is to be used.

Example:

Instead of writing a FØRMAT statement in the following form:

STATEMENT

```
FØRMAT (F6.2,F6.2,F6.2,I5,I5,F8.3,F8.3,F8.3)$
```

it may be written as:

```
FØRMAT (3F6.2,2I5,3F8.3)$
```

## EXPONENT MODIFIER (nP)

The modifier, nP, signifies the $n^{th}$ power of 10 by which the field is to be modified. n, which may be positive or negative, has the effect of scaling the field.

Thus, 3PF2 describing the external fixed-point decimal field 16 would cause the field to be converted to the floating binary equivalent of 16,000 when read into memory. -3PF2 describing the same field will produce the floating binary equivalent of .016. When nP is used with an E output descriptor, the mantissa of the output quantity will be multiplied by $10^n$ and the exponent reduced by n. ($n \geq 0$.) nP has no effect on E input descriptors. The modifier nP will apply to succeeding descriptors (F or E) until 0P is specified.

## MULTI-RECORD CONTROLS

FØRMAT statements can be written to handle coded information in multi-record form, where a record of coded information is defined as a card of input or output or as a printed line of output.

Separate ORDER and FØRMAT statements could be written for each record in order to handle multiple records with different formats. However, a single ALTAC FØRMAT statement can be made to contain all the information necessary for such an operation. Succeeding record formats are separated by slash(/) characters. Repetition of similar formats can be accomplished by using parentheses and repeat modifiers.

Example:

STATEMENT

```
FORMAT (I2, F8.2 / I5, 3F12.6)$
```

If this FØRMAT statement is used in conjunction with an input ORDER statement involving the reading of punched cards, each card would comprise a record and all odd cards would be read according to the first format; all even cards would be read according to the second format specification.

When it is desired to cause repetition of similar fields, a limited parenthetical expression is permitted. A decimal number n appearing to the left of the expression designates the number of times the descriptors and modifiers within parentheses are to be repeated. For example, 2(I2, F3.1, 1PE12.5) is identical to I2, F3.1, 1PE12.5, I2, F3.1, 1PE12.5.

The FØRMAT statement

STATEMENT

```
FØRMAT (2H10I5, 3F8.2/2H102I10, F9.2/ (2H1012F8.2) )$
```

would produce the first two printed lines in the formats I5, 3F8.2 and 2I10, F9.2, and the remaining lines in the format 12F8.2.

In addition, / / will cause a line or record to be skipped, while / / / will produce two blank lines or skip two records.

## ENVIRONMENTAL STATEMENTS

ALTAC Environmental statements do not generate instructions for the resulting machine language program. They are included to furnish the compiler with information needed in defining all input and output media which are used in a program.

The Environmental statements are:

1) IDENTIFY

2) IØUNITS or IØUNITSF

3) CØMPLETE or END

## IDENTIFY STATEMENT

### STATEMENT

Form: IDENTIFY type, mK, nX, nW $

An IDENTIFY statement is included in a program in cases where the Philco 2000 on which the compiled program is to be run differs from the machine which is being used for the compilation process (in core storage size and number of index registers). All parameters in the statement are optional and may be deleted when not required.

Type identifies the program to be of ALTAC form or FORTRAN* form, and may be one of the following two forms:

| TYPE | INTERPRETATION |
|---|---|
| A | ALTAC program |
| F | FORTRAN program |

If FORTRAN and ALTAC statements are not to be mixed within a program, type may be omitted from the IDENTIFY statement, and the compiler will assume that the program is in ALTAC format.

A detailed discussion of the type parameter for FORTRAN programs and for programs in which FORTRAN and ALTAC statements are to be mixed, is presented in Appendix C.

* Automatic coding system used by International Business Machines Corporation

mK defines the size of core storage of the Philco 2000 on which the compiled machine language program will be run, and may be written in one of the following forms:

| mK | SIZE OF OBJECT MACHINE (Words of Core Storage) |
|---|---|
| 8K | 8192 |
| 16K | 16384 |
| 32K | 32768 |

nX defines the maximum number of index registers available in the machine on which the compiled program is to be run, and is written in one of the following forms:

| nX | NUMBER OF INDEX REGISTERS IN OBJECT MACHINE |
|---|---|
| 8X | 8 |
| 16X | 16 |
| 32X | 32 |

nW defines the least number of words of CØMMØN storage which must be contained in the program to be compiled. A discussion of this parameter is presented under CØMMØN statements.

In cases where the same machine, or a machine with the same parameters, is to be used for both the compilation process and running the resulting machine language program, it is not necessary to include an IDENTIFY statement. ALTAC always generates an IDENTIFY statement at the beginning of each main program and subprogram with descriptors K and X. If the programmer does not specify either of these parameters, then ALTAC uses the configuration of the source program computer.

A program that is compiled on a larger machine may run on a smaller machine; however, a program that is compiled on a smaller machine may not run on a larger machine. The programmer is therefore advised to always specify 32K in his IDENTIFY statement.

Example of an IDENTIFY statement:

STATEMENT

IDENTIFY 32K, 8X, 1200W $

(The order in which the parameters appear is insignificant.) The statement specifies that the program is to be run on a Philco 2000 having 32768 words of core storage and eight index registers. At least 1200 words of CØMMØN storage will be reserved.

## IØUNITS STATEMENT*

An IØUNITS statement is used to provide information regarding the input or output units which are to be used during the input or output operation of a program. It contains a complete description of each unit to be used, including such information as record and block sizes. The general form of such a statement is as follows:

| LOCATION | COMMAND | ADDRESS |
|---|---|---|
| | IØUNITS | unit description; unit description; etc. $ |

Only one IØUNITS statement may be in effect during program compilation. If more than one statement is included, only the units described by the first statement will be considered, and all succeeding statements will be ignored. There must be a unit description for each input and/or output unit used by the program.

A unit description, as it applies to each unit, has the following format:

TYPE, UNIT, GROUP SIZE, RECORD SIZE, ERROR ADDRESS, CORE STARTING ADDRESS 1, CORE STARTING ADDRESS 2 (OPTIONAL);

TYPE defines the type of input or output medium. It is one of the following codes:

| TYPE | INTERPRETATION |
|---|---|
| PCu | Punched Cards (output only) |
| PRT | Printer |

* The IØUNITS statement must be in TAC format.

| TYPE | INTERPRETATION |
|---|---|
| DTu | Data Tape (Magnetic) |
| BTu | Binary Tape (Magnetic) |

u specifies the use of the medium. If the medium is to be used only for input, I is written for u; if the medium is to be used only for output, Ø is written for u. If the medium is to be used for both input and output, IØ is written for u. (Only binary tape can be used for both input and output.)

Examples:

| TYPE | INTERPRETATION |
|---|---|
| PCØ | Punched-Card Output |
| BTIØ | Binary Tape Input and Output |
| DTI | Data Tape (Magnetic) Input |

(PCI and DTIØ are non-permissible forms.)

UNIT identifies the particular input-output unit used, and may be any one of the following:

| UNIT | INTERPRETATION |
|---|---|
| nT | The magnetic tape connected to channel n of the Input-Output Processor. (n is any integer 0-16; if n > 16 it will be treated as a symbolic address of a word whose contents (bits 12-15) identifies the particular magnetic tape unit.) |
| SYMBØL | The symbolic address of a word whose contents (bits 12-15) specifies the magnetic tape unit. |

GROUP SIZE is the number of cards in a block or records per block.

RECORD SIZE is the number of computer words per card or computer words per record.

Group and Record Size Parameters are of no effect in an IØUNITS statement describing printed output (DTØ or PRT) or binary tape input or output (BTI, BTØ, BTIØ or BTØI). As shown in the example below, these two parameters would be omitted from the IØUNITS statement with the commas normally separating them retained.

ERROR ADDRESS is a decimal or symbolic address which defines an error jump address which will be executed when an input or output error occurs.

CORE STARTING ADDRESS 1 is the address of the first word of a block of 128 words used for input or output.

CORE STARTING ADDRESS 2 is the address of the first word of a second block for input or output. If this address is present, the program will alternate between areas for reading or writing. If it is not present, or is identical to CORE STARTING ADDRESS 1, no attempt to overlap input and/or output operations with computing will be made. (Binary tape operations require that this second parameter be not specified.)

Example of an IØUNITS statement:

| LOCATION | COMMAND | ADDRESS |
|---|---|---|
| | IØUNITS | DTØ, 12T,,, ERRØUT, BLØCK A, BLØCK B; DTI, 11T, 12, 10, ERRIN, BUFF 1, BUFF 2 $ |

The statement describes data tape output on IOP channel 12 and input of data from IOP channel 11.

## IØUNITSF STATEMENT

The IØUNITSF statement is discussed under ALTAC-FORTRAN Conversion, p. 77.

## CØMPLETE AND END STATEMENTS

The CØMPLETE or END statement is used to signal to the compiler the end of the program being compiled. It is the last physical statement in a program and is written in the following format.

| STATEMENT | STATEMENT |
|---|---|
| CØMPLETE $ | END $ |

Use of either of these statements permits batching of ALTAC programs on the source tape for compilation.

# CHAPTER 7 - ALTAC FUNCTION SUBPROGRAMS AND SUBROUTINES

By writing statements in the ALTAC language presented in the preceding chapters, the reader can direct the Philco 2000 in the performance of practically any type of numerical calculation. Information may be transferred between internal storage and an external medium in a variety of formats. Decisions and paths of flow may be set up through the use of control statements. Existing TAC subroutines and other functions are readily available, as well as functions incorporated on the TAC library tape.

The reader has seen in Chapter 3 that these functions can be called upon by writing ALTAC statements containing the names of the functions and lists of arguments enclosed within parentheses. Functions of his own creation, not available from these sources, may be defined by a single statement preceding the first executable statement in the program. Such functions are then referred to in the same manner as any of the other types.

However, not all functions can be completely defined within the limit of a single ALTAC statement. Furthermore, in most cases it would be inadvisable to include them on the TAC library tape due to their limited application. It would likewise be inconvenient and inefficient to write them as a sequence of statements in the main program, repeated each time they are needed.

A different concept of function definition is presented in this chapter which will enable the reader to define such functions by writing separate programs, and compiling and checking them apart from the main program. Such programs are referred to as subprograms, since they are designed to operate under control of other programs. A subprogram may in itself constitute a calling program, utilizing other functions or subprograms. Existing programs may readily be converted into subprograms, and the process expanded indefinitely within the limits of the capacity of the Philco 2000.

A second type of ALTAC subprogram can be written as a subroutine. While functions are designed primarily to perform intermediate calculations producing single valued results, such as square root, summations, etc., subroutines are more suitable for other purposes. They may be used to perform a complete series of calculations, the results of which are stored in memory locations at program run time.

Linkage of subprograms of either type to referencing or calling programs can be accomplished by means of five ALTAC statements, namely:

1) CALL

2) SUBRØUTINE

3) RETURN

4) FUNCTIØN

5) END

While functions are referred to in much the same manner as functions of the types described earlier, subroutine subprograms must be referred to by a CALL statement included as a separate statement in the calling program. A FUNCTIØN statement must be the first statement identifying a subprogram of the function type, while a subroutine subprogram must always be headed by a SUBRØUTINE statement. The last logical statement of each type of subprogram must be a RETURN statement, which passes control back to the calling program. The form in which these statements must be written, along with their components, will be discussed in detail in the sections that follow, along with appropriate illustrations.

CALL

Form: CALL subroutine name (arguments) $

Subroutine Name is the symbolic name used to refer to a subroutine. It is composed of from one to seven alphanumeric characters, the first of which must be alphabetic.

Arguments are parameters presented to the subroutine to be operated upon. They are separated by commas, enclosed in parentheses, and may be in any of the following forms:

1) Fixed-or floating-point constants.

2) Fixed-or floating-point variables or subscripted dimensioned variables.

3) Names of arrays with subscripts omitted.

4) Any ALTAC arithmetic expression.

5) Hollerith characters.

The use of Hollerith characters (alphanumeric, special and space) as an argument of a function is presented below. The other types of arguments listed above were discussed on pp. 15 - 18.

Hollerith arguments may be used for many purposes. For example, table look-up, type-out of comments, and so on. If the called subroutine

is in ALTAC or FORTRAN language, the dummy variable in the subroutine, corresponding to the Hollerith argument in the calling statement, must be singly dimensioned.

The Hollerith argument in the calling statement must be of the following general form:

nH.....

where n is any decimal integer greater than zero. The n characters following the H will be translated by ALTAC into TAC word constants (W/.....), eight characters per word. If n is not a multiple of eight, the unfilled part of the last word will be filled with spaces. A sentinel of 48 ones will follow the last word.

What is actually transmitted to the subroutine is a word containing the starting location of the Hollerith information in its left address. All this information is generated by ALTAC as pool constants (with a P in the label field).

Example:

```
    CALL      SUBR (10HΔEXAMPLEΔ1)
```

Generated coding:

```
              TMA   0001HØL          $
S             SUBR                  $
P 0001HØL     C/HLT, 0002HØL        $
P 0002HØL     W/ΔEXAMPLE            $
P             W/Δ1ΔΔΔΔΔΔ            $
              48/1                  $
```

The CALL statement, when written in the calling program, produces a jump to a subroutine subprogram at program execution time. To ensure compatibility, the arguments in the CALL statement must be presented in the same order, number, form and mode as the corresponding arguments in the SUBRØUTINE statement written at the beginning of the subprogram being called.

## SUBRØUTINE

Form: SUBRØUTINE name (arguments) $

Name is the symbolic name of the subroutine. The rules for its formation are the same as described above for the CALL statement.

Arguments are variable names without subscripts which are used in the subroutine. If they represent arrays, a DIMENSIØN statement must be written in the subprogram as well as in the calling program.

The arguments are separated by commas and are enclosed in parentheses. The actual variable names used as arguments in a SUBRØUTINE statement need not necessarily be the same as the corresponding variable names written as arguments in the CALL statement, as long as they agree in order, number, and mode. If a single element of an array is presented as a subscripted variable in a CALL statement argument, the array must not be dimensioned in the subroutine.

Example of a Subroutine Subprogram:

Assume A and B are two single-dimensioned arrays of 100 elements each. A third array C is to be formed as a result of the following calculations involving A and B:

C = A - B if A > B
C = B - A if A < B
C = 0 if A = 0 and/or B = 0
C = 0 if A = B

The main calling program might appear as follows:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | DIMENSIØN A (100), B(100), C(100) $ |
| | Program |
| | Statements |
| 5 | CALL CALC (A, B, C) $ |
| 6 | Some Statement |
| | |
| | |

The subroutine thus referred to could be written in the following manner:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | SUBRØUTINE CALC (S, T, U ) $ |
| | DIMENSIØN S (100), T (100), U (100) $ |
| | DØ (END) I = 1, 100 $ |
| END | IF (S(I) ) E (0), U(I) = 0; IF (T (I) ) E (0), U(I) = 0; |
| | IF (S(I) ) E (T(I) ), U(I) = 0.; |
| | IF (S(I))GT (T(I) ), U(I) = S(I) - T(I); IF (S(I) ) LT (T(I) ), |
| | U(I) = T(I) - S(I) $ |
| 10 | RETURN $ |
| | END $ |
| | |
| | |

Statement 5 in the main program jumps to the subroutine CALC. Notice that the arrays S, T, and U used in the subroutine, actually represent arrays A, B, and C of the same mode (floating point), order, and number. Statement 10 of the subroutine produces a jump back to statement 6 of the main program, at which time the resulting array C will have been formed.

RETURN

Form: RETURN $

This statement completes a function or subroutine subprogram, and produces a jump back to the calling program, as in the above example. It must be the last logical step in a subroutine or function subprogram.

FUNCTIØN

Form: FUNCTIØN name (arguments) $

Name is the symbolic name of the function subprogram. It is composed of from one to seven alphanumeric characters, the first of which must be alphabetic. If the first character is I, J, K, L, M, or N the value of the function will be in fixed point. Otherwise the value of the function will be in floating point.

Arguments must be written subject to the same rules as for subroutine arguments. A function subprogram is called for in the same manner as other types of functions - by writing an arithmetic statement including the function name along with a list of arguments enclosed in parentheses. Thus, the statement Y = A + B-SUM(T) + C/D $ would call the function subprogram headed by the statement: FUNCTIØN SUM (ELEMENT) $.

For example, if T is of maximum dimension, 250, in the main program, the function subprogram would appear as follows:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | FUNCTIØN SUM (ELEMENT) $ |
| | DIMENSIØN ELEMENT (250) $ |
| | SUM = ELEMENT (1) $ |
| | DØ 9 I = 2, 250 $ |
| 9 | SUM= SUM + ELEMENT (I) $ |
| 10 | RETURN $ |
| | END $ |

Statement 10 would cause a jump back to the same statement in the calling program from which exit was made, and the calculation would continue. In a function subprogram, the name of the function must appear as a variable to be evaluated on the left side of an arithmetic statement, as in statement 9 in the example.

END

Form: END $

Description:

An END statement is used to separate subroutine and function subprograms from each other and from the main program with which they are being compiled. The END statement must appear as the last physical statement of a subroutine or function subprogram.

# CHAPTER 8 - ADDITIONAL FEATURES OF THE ALTAC SYSTEM

This chapter is devoted to the discussion of TAC coding within an ALTAC program, the Format Control Cards, BITSBITSBITSBITS and TACLTACLTACLTACL, and the I Card.

## TAC CODING WITHIN AN ALTAC PROGRAM

TAC coding in the standard TAC format may be included in an ALTAC program in either of two ways:

1) By writing the statement,

STATEMENT

STARTTAC $

followed by the TAC coding, and terminated by the ALTAC instruction

| COMMAND | ADDRESS |
|---|---|
| ENDTAC | $ |

All coding between these statements is, at this point, included in the TAC program that results from the ALTAC translation.

2) By punching a T in the label field (column 9 for ALTAC programs and column 1 for FORTRAN programs) of every TAC insert. ALTAC replaces the T in the label field of an ALTAC card with a space character. It then interprets columns 9-80 literally, as it does with a FORTRAN card.

A card with a T in the label field must never appear between the statements STARTTAC and ENDTAC.

Within the TAC portion of the program, reference may be made to variable names used in the ALTAC section. The use of TAC coding to define arrays is discussed under TABLEDEF statements.

## FORMAT CONTROL CARDS

If the source program language is mixed, i.e., TAC language cards with Binary Relocatable cards, proper control cards (BITSBITSBITSBITS and TACLTACLTACLTACL) must be used to identify the type of source language that follows.

## BITSBITSBITSBITS

A BITSBITSBITSBITS card is placed immediately before a binary input deck to indicate to the compiler that binary input follows. If binary relocatable coding with its preceding BITSBITSBITSBITS card follows any card other than the I Card or other BITSBITSBITSBITS cards, ALTAC generates an ILLEGAL BITSBITSBITSBITS remarks card and terminates compilation immediately. BITSBITSBITSBITS is punched in Hollerith, in columns 9-24 of a card.

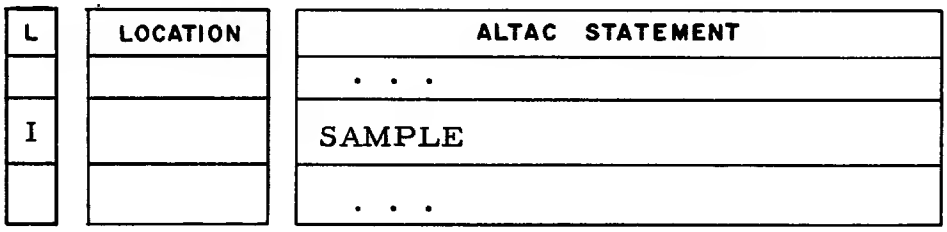
## TACLTACLTACLTACL

A TACLTACLTACLTACL card must be placed immediately before a TAC language deck (1) to indicate to the compiler that TAC language follows, or (2) to indicate a change in mode (Image to Code or Code to Image) of the TAC language deck. TACLTACLTACLTACL is punched in Hollerith, in columns 9-24 of a card.

## I CARD

The I Card is usually the first physical card of a program, and it serves to identify the program.

Information on the I Card must be written as follows:




An I is written in the label field (column 9), while a name (e.g., SAMPLE) identifying the program is written starting at column 17. The name can comprise as many as 16 characters and is not terminated with a $ character.

At least one non-zero character must appear in columns 17-24 of the I Card, otherwise ALTAC will not include the I Card in the generated TAC coding.

ALTAC permits the I Card to precede a BITSBITSBITSBITS card and any number of binary relocatable cards, provided no card precedes the I Card other than any number of optional TACLTACLTACLTACL cards. If no BITSBITSBITSBITS card follows the I Card, ALTAC permits any number of blank cards (as well as TACLTACLTACLTACL cards) to precede the I Card.

# APPENDIX A

## EXERCISES

Translate the following arithmetic expressions into acceptable ALTAC statements:

1. $R = \frac{100A}{B}$

2. $\frac{100R5}{R4} = R8$

3. $TEM = \cdot\frac{6838(DATA - RDATA)}{RDATA2 + 459.7}$

4. $OUT4 = \frac{OUT9 - (TEM3)ABSOER1 + TEM4}{TEM5} \quad (159,000)$

5. $OUT3 = 543(A(B \cdot W^2 + C \cdot W^3) + TEM) + \frac{RUN}{HOUR} + DATA(KURV + .01)$

6. $ROOT = \frac{(3.415) - 4A(6.54)}{2A^2} + C^2 + \frac{D^3}{F^4} \quad (5A + 3.415)$

7. $X = (1,000,000A + 5,000B + 40C) \div (2W^2 + 10.598)$

8. $B_i \div C_i = A_i$

9. $OUT4 = OUT_i - \frac{(TEM3)ABSOER_i(159,000,000)}{TEM5}$

10. $RUN = A^{2c} - B^{-2} + 5EAT_i((A^{2c} + B^{-2}) + 3.1416)$

11. Given the problem:

    $1268X - 1195Y = 421$

    $1395X + 1003Y = 721$

    Output is to be on Data Tape (Magnetic)

    Evaluate X and Y
    for later printing.

12. Given values for A, B, C, D and 25 values of x. Evaluate the function

$$F(x) = \begin{cases} Ax^2 + Bx + C & \text{if x is less than D} \\ 0 & \text{if x equals D} \\ -Ax^2 + Bx - C & \text{if x is greater than D} \end{cases}$$

Assume input data is punched on cards as follows:

CARD 1: A SXXX.XXX Columns 1 - 8
B SXXX.XXX Columns 9 - 16
C SXXX.XXX Columns 17 - 24
D SXXX.XXX Columns 25 - 32

CARDS 2 - 4 with values of x punched SXXX.XXX ten values per card. The last value will be punched in columns 33 - 40 of card number 4.

Write each value of x, F(x), on a data tape (magnetic) along with heading identification.

13. Find the values of the variables by 10 successive iterations

$$5.32X1 + 1.20X2 + .22X3 - 1.60X4 = 7.32$$

$$-2.01X1 + 4.21X2 + 3.12X3 + 1.01X4 = 6.23$$

$$2.52X1 - 1.22X2 + 6.05X3 + 3.12X4 = 4.16$$

$$3.18X1 + 0.99X2 - 0.13X3 + 7.66X4 = 5.23$$

Method:

Begin by setting each X equal to zero. Solve the first equation for X1; the second for X2; the third for X3; and the fourth for X4.

14. Given: $a_i$, $b_i$, $c_i$ for i = 1,....., 100, compute

$$\text{RESULT} = \left( \sum_{i=1}^{100} (a_i c_i)^2 \right) \left( \sum_{i=1}^{100} (b_i - c_i) \right) \Big/ \left( \sum_{i=1}^{100} (a_i b_i - c_i^2) \right)$$

15. Find the area of a triangle by using the following formulas:

$$\text{AREA} = \sqrt{S(S-A)\ (S-B)\ (S-C)}$$

$$S = \frac{1}{2}\ (A + B + C)$$

Restrictions:

The square root function is not to be used.

Output:

A, B, C, S, and AREA

# APPENDIX B

## TABLE OF ALTAC CHARACTERS

| Character | Card | PHILCO 2000 And Tape (Octal) | Character | Card | PHILCO 2000 And Tape (Octal) | Character | Card | PHILCO 2000 And Tape (Octal) |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 00 | D | 12-4 | 24 | Q | 11-8 | 50 |
| 1 | 1 | 01 | E | 12-5 | 25 | R | 11-9 | 51 |
| 2 | 2 | 02 | F | 12-6 | 26 | - | 11 | 40 |
| 3 | 3 | 03 | G | 12-7 | 27 | $ | 11-8-3 | 53 |
| 4 | 4 | 04 | H | 12-8 | 30 | * | 11-8-4 | 54 |
| 5 | 5 | 05 | I | 12-9 | 31 | / | 0-1 | 61 |
| 6 | 6 | 06 | + | 12 | 20 | S | 0-2 | 62 |
| 7 | 7 | 07 | . | 12-8-3 | 33 | T | 0-3 | 63 |
| 8 | 8 | 10 | ) | 12-8-4 | 34 | U | 0-4 | 64 |
| 9 | 9 | 11 | J | 11-1 | 41 | V | 0-5 | 65 |
| Blank | | 60 | K | 11-2 | 42 | W | 0-6 | 66 |
| = | 8-3 | 13 | L | 11-3 | 43 | X | 0-7 | 67 |
| ; | 8-4 | 14 | M | 11-4 | 44 | Y | 0-8 | 70 |
| A | 12-1 | 21 | N | 11-5 | 45 | Z | 0-9 | 71 |
| B | 12-2 | 22 | Ø | 11-6 | 46 | , | 0-8-3 | 73 |
| C | 12-3 | 23 | P | 11-7 | 47 | ( | 0-8-4 | 74 |

## APPENDIX C

## PREPARATION OF FORTRAN II DECKS FOR ALTAC COMPILATION

Programs to be compiled by the ALTAC compiler may be in either ALTAC or FORTRAN format. For programs in FORTRAN format, the following procedures must be followed in preparing them for ALTAC compilation.

### I CARD

An Identification (I) card is the first physical card of a deck, and it must be prepared for each program. With the REL option specified, a program is considered to be a single subroutine or main program preceded by any number of binary relocatable subroutines. (Each subroutine of a REL program, therefore, must have an I Card.) With the RPL or ABS option, a program would be comprised of any number of subroutines preceding a main program (which must be the last physical deck of cards). The I Card is further discussed on p. 69.

### IDENTIFY STATEMENT

Programs in ALTAC and FORTRAN format may be mixed in any order, provided the proper IDENTIFY cards are included. An IDENTIFY statement (illustrated below) must appear ahead of all statements in FORTRAN format, even before FORTRAN remarks, since ALTAC assumes ALTAC format unless otherwise instructed.

| LOCATION | ALTAC STATEMENT |
|---|---|
| | |
| | . . . |
| | IDENTIFY type, mK, nX, nW $ |
| | . . . |
| | . . . |
| | |

This statement must start in column 17 or beyond, and is terminated with a $ character.

| Parameter | Explanation |
|---|---|
| type | Type must be F or A. If FORTRAN and ALTAC statements are to be mixed within a program, an IDENTIFY statement with the appropriate type (A or F) must precede each change. An IDENTIFY F indicates that FORTRAN statements follow, while an IDENTIFY A would indicate that an ALTAC insert (in a FORTRAN program) follows. The other parameters, except type, in an IDENTIFY statement need only be specified initially. |
| mK | This parameter defines the size of core storage of the Philco 2000 System on which the compiled program is expected to be run. (See page 58.) mK is optional. |
| nX | This parameter indicates the number of index registers available with the Philco 2000 System on which the compiled program will be run. (See page 58.) If a program is to be run on the same size (i.e., same size of core and the same number of index registers) machine as that on which it was compiled, these parameters mK and nX need not be specified. |
| nW | This parameter states that the program compiled must contain at least n words of CØMMØN storage. (See page 40.) This parameter is optional, and is to be included only when compiling with the RPL or ABS option. |

Example:

| LOCATION | ALTAC STATEMENT |
|---|---|
| | |
| | . . . |
| | IDENTIFY 8K, 8X, 1200 W, F $ |
| | . . . |
| | |

As previously mentioned, the IDENTIFY statement must start in column 17, except when switching from F back to A format, in which case, the IDENTIFY may be written starting at column 7. (As shown above, the order of the parameters is insignificant.)

According to the above statement, the ALTAC compiler will assume the deck to be of FORTRAN format as soon as the F is encountered. It will prepare the program to be run on a Philco 2000 having 8192 words of core storage and eight index registers.

Assuming RPL or ABS option, at least 1200 words of CØMMØN storage will be reserved, provided the first subprogram does not require more. Assuming REL option, common storage is reserved automatically by a CØMSTØR card for each common variable.

## SEQUENCE OF EQUIVALENCE, CØMMØN, AND DIMENSIØN STATEMENTS

Before an array is used in a main program or in a subprogram, it must be defined in the DIMENSIØN statement. In that same program or subprogram, all EQUIVALENCE statements should appear first, followed by all CØMMØN statements, followed by all DIMENSIØN statements. These should all appear before the use of any subscripted variables.

## IØUNITSF STATEMENT

It was shown that for ALTAC programs an IØUNITS statement is written in ALTAC format, starting at column 17 or beyond. For FORTRAN programs, IØUNITS is written in FORTRAN format, as IØUNITSF starting at column 7 or beyond. IØUNITSF has the same parameter format as the IØUNITS statement.

An IØUNITS statement produces coding which assumes that the programmer is supplying both v (vertical format) and s (data select) control characters for every line that is to be printed; IØUNITSF automatically produces an s character of 0 and makes the following changes in the v character:

| PRINTER PROGRAM-CONTROL CHARACTER | | V | S | MEANING |
|---|---|---|---|---|
| 0 | becomes | 1 or 1 | 0 or 0 | Double space |
| Δ (a space symbol) | becomes | 1 | 0 | Single space |
| 1 | becomes | 7 | 0 | Skip to top of page |
| + | becomes | 0 | 0 | No space |
| A filler character | becomes | 1 | 0 | Single space |
| Anything else | becomes | 7 | 0 | Skip to top of page |

If the IØUNITSF statement is written according to FORTRAN conventions, it must follow the IDENTIFY statement. If, on the other hand, IØUNITSF is written according to ALTAC conventions and starts at column 17 or beyond, it must precede the IDENTIFY statement.

| ALTAC STATEMENT |
|---|
| . . . |
| IØUNITSF unit description; unit description; - - - $ |
| . . . |
| . . . |

The parameters of this statement appear in the same order as outlined on p. 59 of this manual, and the statement is written in the same form as any other FORTRAN or ALTAC statement.

All "on-line" units (card reader, card punch, and printer) must be described as magnetic tape units in the IØUNITSF statement in the following manner:

(a) For all READ orders, specify 11T for the Unit Parameter and DTI as the Type Parameter.

(b) For all PRINT orders, specify 12T for the Unit Parameter and DTØ as the Type Parameter.

(c) For all PUNCH orders, specify 13T for the Unit Parameter and PCØ as the Type Parameter.

Tape units 11T, 12T, and 13T are used to avoid conflict with logical tapes 1-10 which are used in FORTRAN programs for the IBM 704 computers.

The READ statement is treated as if it were a READ INPUT TAPE 11 instruction. The IØUNITSF statement should therefore contain an entry which starts with DTI, 11T, 12, 10,..... It is, of course, necessary to convert the cards off line to magnetic tape in Code Mode, 10 words per card, 12 cards per block.

Similarly, when a PRINT statement is used, the IØUNITSF statement should contain DTØ, 12T,.... When a PUNCH statement is used, it should contain PCØ, 13T....

Error addresses, if specified, must be symbolic. Normally, they are not specified, in which case the PRØC error exit, which is a standard error return, is used.

Only one IØUNITSF statement must be given with a complete program.

## FØRMAT STATEMENTS

The present version of ALTAC requires the following changes in the FORTRAN program in order to handle Hollerith input and produce compatible results. Input FØRMAT statements must be checked to determine if Hollerith input (under the nH descriptor) is specified. This is not permitted in ALTAC and, if found, must be corrected by performing the following:

(a) Symbolic names must be assigned to locations into which the Hollerith information will be read. These symbols must then be inserted in correct sequence into the LIST of the READ statement, and must also be appropriately added to the DIMENSIØN statement, if required.

(b) A new FØRMAT statement must be prepared using the Aw descriptor in place of the nH descriptor. A single Aw descriptor must describe no more than eight characters of input. Successive Hollerith fields of the same width can be described by modifying the Aw descriptor with a repeat modifier. Each Philco 2000 word may contain a maximum of eight alphanumeric characters. Where less than eight are specified, as in A4, the characters are left justified in a word in memory, and the rest of the word is filled with blanks. Where more than eight alphanumeric characters are specified, for example, A12, the last (w-8) characters will be lost. In replacing nH, therefore, the following is done:

(1) Divide n by 8.

(2) If the division is exact, prefix the quotient to A8.

(3) If the division is inexact, prefix the integer part of the quotient to A8; and suffix the remainder to another A immediately following the previous.

For example, 20H would become 2A8, A4; 14H would become A8, A6; and 5H would become A5. In the first example, it would also be necessary to expand the LIST in the READ statement (which would receive the Hollerith information) by three memory locations; in the second example, two memory locations must be inserted into the LIST; and in the third example, an insertion of one memory location is necessary.

It is also necessary to include a DIMENSIØN statement, which sets up the area into which information is read, and to change the corresponding READ statement to read into this area. Thus, the FORTRAN statements

```
      READ 15
        .
        .
15    FØRMAT    (20H.....)
```

should be changed to

```
      DIMENSIØN HØLLI (3)
        .
        .
      READ 15, HØLLI

15    FØRMAT (2A8, A4)
```

Because Aw is acceptable to both FORTRAN and ALTAC, its use is recommended in preference to Ww.

Commas must always separate descriptors. (For example, FØRMAT (5X9H.....) is acceptable to FORTRAN, but should be changed to FØRMAT (5X, 9H.....) for proper ALTAC interpretation.)

## CØMPLETE AND END STATEMENTS

CØMPLETE and END statements are used interchangeably in ALTAC. END must always be followed by a dollar sign or a left parenthesis. These statements are like any other FORTRAN statement. In any program, (FORTRAN or ALTAC), never use END or CØMPLETE in a compound statement in which it is not the last component.

## ARRAYS

All multi-dimensional arrays must be referred to by proper subscripts. The number of subscripts appending a variable must match the number of dimensions specified in the DIMENSIØN statement:

Example 1:

| ALTAC STATEMENT |
|---|
| . . . |
| DIMENSIØN A(8, 8) |
| . . . |
| B = A(9) |
| . . . |

The above form is not permitted in ALTAC. Instead, A(9) must be written as A(1,2) so that the subscripts agree in number with the subscripts in the DIMENSIØN statement.

Example 2:

| ALTAC STATEMENT |
|---|
| . . . |
| SUBRØUTINE SUB (A) |
| . . . |
| DIMENSIØN A(8) |
| . . . |
| B = A |
| . . . |

Because in ALTAC, A is not always the same as A(1), the above form (B = A) is not permitted. A, appearing alone, in this case designates the address of the first word of the array A, while A(1) always designates the contents of the first word of the array.

## SAP CODING

Delete all SAP coding which may appear in the program and replace with either TAC coding or FORTRAN statements.

The first statement in an ALTAC (or FORTRAN) main program should not be a TAC insert, since, during compilation, ALTAC generates the program's END-START transfer address upon encountering the first ALTAC (FORTRAN) statement.

## ASSIGNED GØ TØ STATEMENTS

The restrictions in FORTRAN on ASSIGNED GØ TØ's are not applicable. If the address of a GØ TØ statement or of an IF statement appearing in the range of a DØ, is not defined in that range, ALTAC assumes that a jump is going to be made out of the range of the DØ. Similarly, if the address of an ASSIGNED GØ TØ is not listed, it would appear to ALTAC as though a jump were being made outside the range of the DØ.

## STØP AND END FILE STATEMENTS

All ALTAC/FORTRAN programs must contain at least one STØP or one END FILE statement to produce the proper RUNØUT of output information, or it can include the coding:

```
STARTTAC  $
RUNØUT    $
END TAC   $
```

A STØP statement is the only ALTAC statement that provides proper RUNØUT of all stored output data.

## FLOATING-POINT ZERO

Since floating-point zero and fixed-point zero are not the same in the Philco 2000, the assumption should not be made that memory has been pre-cleared to floating-point zero. For example, when testing the contents of a given location (A) for zero, if A has not been defined previous to the test, the statement A = 0. should precede any such test. For greater familiarity with floating-point operations in the Philco 2000, the reader is referred to Appendix E of the Programming Manual.

## DIVISION BY ZERO

Note where division by zero is attempted. Mathematically, division of a finite number by zero yields infinity. ALTAC therefore yields the largest positive floating-point number possible:

$$3777777777773777_{(8)}$$

Division by zero in most versions of FORTRAN, however, yields zero. Although such a result is mathematically incorrect, to achieve compatibility with FORTRAN (in programs using this practice), it is possible to change the constant Ø/3777777777773777 in the subroutine IBG to F/0. In ALTAC, zero is always positive. Negative zero cannot be represented in the Philco 2000.

## SUBROUTINES

Note that if the computer jumps to memory location 3(1SUBERR) an error has occurred upon executing a subroutine. For example, the computer will jump to location 3 if the square root (SQRTF) subroutine is entered with a negative argument. FORTRAN in such a case takes the square root of the absolute value of the argument, ALTAC does not.

## SENSE SWITCH STATEMENTS

The programmer must provide information as to the proper setting of console toggle switches when IF SENSE SWITCH statements are included in the program. Toggles 1-6 refer to SENSE SWITCHES 1-6.

## PAUSE STATEMENTS

A PAUSE statement is usually included in a program to provide time for operator intervention. In such a case, the programmer must specify the action to be taken. ALTAC does not produce a runout of the output buffer areas for a PAUSE statement. (The reader is referred to the IØPS Manual for a discussion of RUNØUT.)

## ENDING A PROGRAM

Programs that end on a card reader select or on an end of file condition from an input tape, must be modified to be able to recognize when there is no more data to be processed; and, at this point, to transfer control to a STØP statement or equivalent TAC coding, to runout partially filled buffers. A program should not be ended on a read-write-check error.

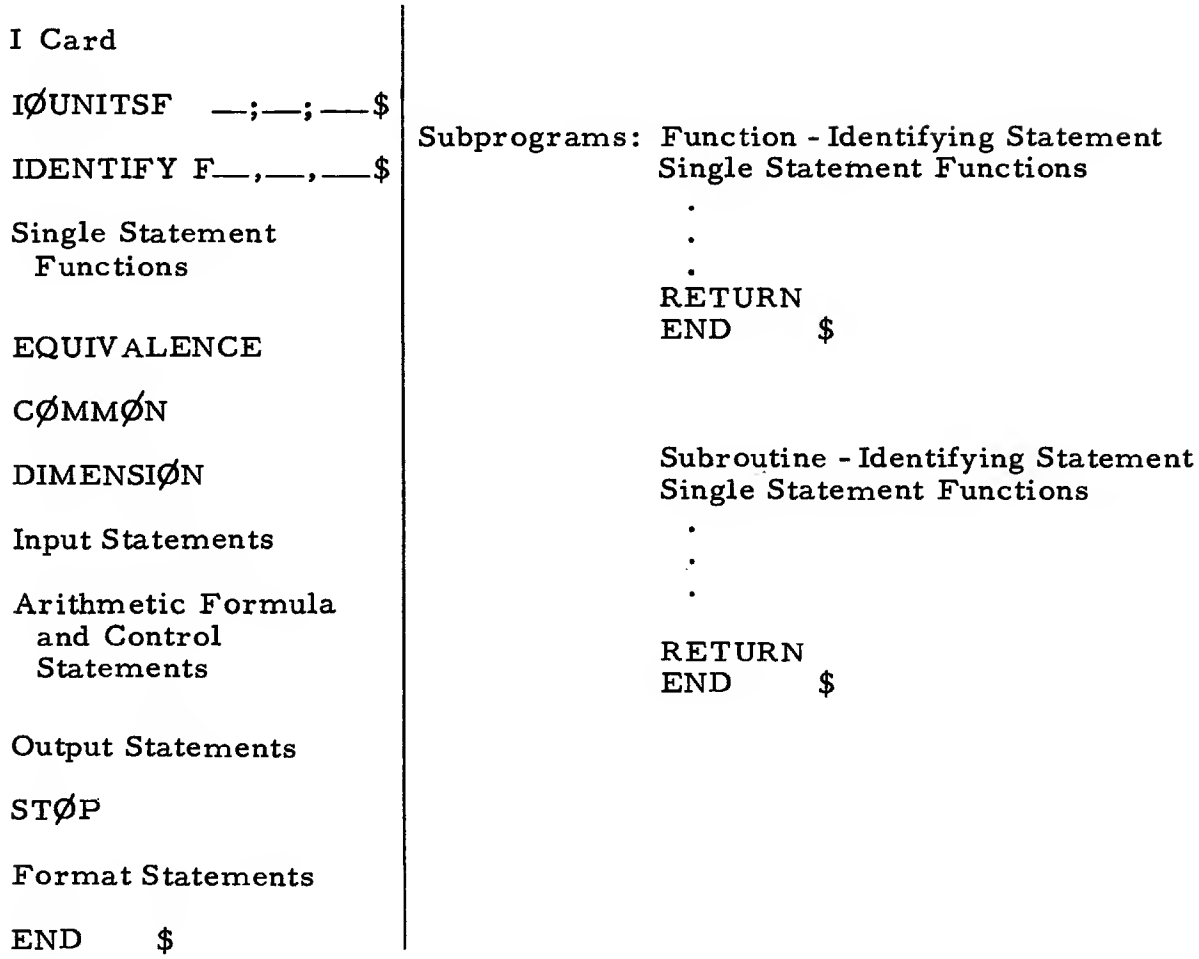
## BINARY TAPE OPERATIONS

Group and Record Size parameters need not be specified in an IØUNITSF statement for binary tape operations. IØPS has its own fixed group and record size for binary tape operations and will ignore any such specification, if written. Specifying two buffer blocks (double buffering) is not permitted with binary tape operations; only one buffer block should be specified.

When writing the binary tape order statement, remember that each order will cause at least one full block to be read from or written on tape. The larger the list of binary information to be transferred, the more efficient the execution of the pertinent section of the IØPS program will be.

Note also that there is no restriction on the complexity of the LIST structure for binary tape orders with ALTAC since input-output operations are buffered.

## BASIC ARRANGEMENT OF A FORTRAN PROGRAM FOR ALTAC COMPILATION




The same physical order of statements holds for both main- and subprograms with the exception of the environmental cards I, IDENTIFY and IØUNITSF. Only when the REL option is specified must subprograms have their own Identification (I) and IDENTIFY cards.

Single statement functions should precede EQUIVALENCE, CØMMØN, and DIMENSIØN statements.

## CHECK SHEET FOR ALTAC-FORTRAN COMPILATIONS

1. An Identification (I) card should be the first card of the deck.

2. Prepare an IDENTIFY F statement.

3. Prepare an IØUNITSF statement. All "on-line" operations involving peripheral equipment (card reader, card punch, and printer) must be described as magnetic tape operations involving tapes 11, 13, and 12 respectively.

4. Subprograms must appear before the main program under RPL or ABS options.

5. EQUIVALENCE, CØMMØN, and DIMENSIØN statements must be arranged in this order, and must appear before any reference to an array.

6. FORTRAN-type END statements must terminate all SUBRØUTINE and FUNCTIØN subprograms. END should be followed by a $ character.

7. All multi-dimensional arrays must be referred to by proper subscripts.

8. Delete all SAP coding and replace with equivalent TAC coding.

9. Boolean Algebra calculations must be done in TAC coding; ALTAC does not offer this provision at this time.

10. Never assume that any variable is preset by ALTAC. Always include all such necessary "housekeeping" in the source program.

11. When Hollerith input is specified, the associated field descriptors (nH) must be changed to Aw descriptors.

12. STØP or ENDFILE statements should be used at the logical end of a program to cause proper runout of output information. Do not end a program with a read-write check error.

13. In FORTRAN, division of a finite number by zero yields zero. In ALTAC, the result would be $3777777777773777_{(8)}$. Check to determine if this difference is significant to the successful running of the program. Based on this decision, the constant $3777777777773777_{(8)}$ in the subroutine IBG should or should not be changed to F/0.

## APPENDIX D

## POSSIBLE GROUPING OF CHARACTERS $\alpha\beta$ IN ALTAC FØRMAT STATEMENTS

$\alpha$ = preceding character

$\beta$ = succeeding character

Y = YES (permitted)

N = NO (not permitted)

| $\alpha$ \ $\beta$ | n | , | $ | / | . | ( | ) | + | - | I | F | E | A | W | X | H | R | P |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| n | Y | Y | Y | Y | Y | Y | Y | N | N | Y | Y | Y | Y | Y | Y | Y | Y | Y |
| , | Y | N | N | Y | N | N | N | Y | Y | Y | Y | Y | Y | Y | Y | N | N | N |
| $ | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N |
| / | Y | N | Y | Y | N | N | Y | Y | Y | Y | Y | Y | Y | Y | Y | N | N | N |
| . | Y | Y | Y | Y | N | N | Y | N | N | N | N | N | N | N | N | N | N | N |
| ( | Y | N | N | Y | N | N | N | Y | Y | Y | Y | Y | Y | Y | Y | N | N | N |
| ) | N | Y | Y | Y | N | N | N | N | N | N | N | N | N | N | N | N | N | N |
| + | Y | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N |
| - | Y | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N |
| I | Y | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N |
| F | Y | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N |
| E | Y | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N |
| A | Y | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N |
| W | Y | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N |
| X | Y | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N |
| H(+n) | N | Y | Y | Y | N | N | Y | N | N | N | N | N | N | N | N | N | N | N |
| R | Y | N | N | N | N | N | N | Y | Y | Y | Y | Y | Y | Y | Y | N | N | N |
| P | Y | N | N | N | N | N | N | N | N | Y | Y | Y | N | N | N | N | N | N |

# APPENDIX E

## ALTAC TABLE SPECIFICATIONS

The following table lists, according to machine size, the maximum allowable number of certain elements of a program:

| | SIZE OF MACHINE | | |
|---|---|---|---|
| | 8,192 WORDS | 16,384 WORDS | 32,768 WORDS |
| MAXIMUM SIZE OF TEMPORARY STORAGE AREA | 257 | 1500 | 2000 |
| MAXIMUM NUMBER OF DØ's IN A PROGRAM | 90 | 200 | 200 |
| MAXIMUM NUMBER OF DØ's IN A NEST | 63 | 63 | 63 |
| MAXIMUM NUMBER OF FØRMAT STATEMENTS IN A PROGRAM | 125 | 1000 | 1000 |
| The actual number of words in a DIMENSIØN Table is approximately equal to (2m + 3n), where m represents the total number of one-dimensional arrays appearing in DIMENSIØN statements that do not appear in EQUIVALENCE and CØMMØN statements, nor as arguments of a subroutine or function subprogram. n represents all other arrays. The quantity (2m + 3n) should not exceed the MAXIMUM ALLOWABLE NUMBER OF WORDS IN THE DIMENSIØN TABLE | 480 | 1000 | 1000 |
| MAXIMUM NUMBER OF VARIABLES IN EQUIVALENCE OR CØMMØN STATEMENTS | 258 | 750 | 1500 |
| MAXIMUM NUMBER OF ARGUMENTS IN A SUBROUTINE OR FUNCTION SUBPROGRAM | 31 | 255 | 255 |

# APPENDIX F

## ALTAC DIAGNOSTICS

The following are some of the diagnostics which are generated by ALTAC and which will appear on the code-edit as a result of the compilation run.

A. ILLEGAL STATEMENT is printed when

1. An unequal number of left and right parentheses is contained in a single statement.

2. An illegal character appears in a statement.

   Example:

   D0 12 I = 1, 25$
   where the Ø intended is punched as a zero.

B. ILLEGAL LOCATION is printed when

One of the special characters appears in a location field or when an alphabetic character appears in a location field whose first non-space character is numeric.

C. AMBIGUOUS FLAD is printed when

Two or more subroutines in the same compilation have the same name.

D. TOO MANY ARRAYS is printed when

Too many storage locations are used for arrays. There is not enough room in the ALTAC program to handle all arrays. (ALTAC does not overlap memory.)

E. TOO MANY UNKNOWNS

This diagnostic will appear if the following conditions are present:

1. There are more than 31 arguments in a single subroutine in the 8K version of ALTAC.

2. There are more than 255 arguments in a single subroutine in the 16K or 32K version of ALTAC.

F. TOO MANY WORDS is printed when

The size of the core specified in the IDENTIFY statement is exceeded by the variables listed in a DIMENSIØN statement.

G. TOO MANY EQ./COM. is printed when

Too many EQUIVALENCE and/or CØMMØN locations are specified and the respective lists are exceeded.

H. TOO MANY DO LOOPS is printed when

The number of DØ's in a program or in a nest exceeds the maximum allowable number. These limitations are indicated in the table in Appendix E.

I. UNENDED DO is printed when

The last statement in the range of a DØ is not present in the source program. One or more UNENDED DØ's results in an ILLEGAL DØ NEST diagnostic.

J. TOO MANY FORMATS is printed when

The number of FØRMAT statements in a program exceeds the maximum allowable number indicated in Appendix E.

K. ILLEGAL BITSBITSBITSBITS is printed when

Binary relocatable coding follows any card other than I or BITSBITSBITSBITS.

# INDEX

# INDEX

# PHILCO

*Famous for Quality the World Over*

A SUBSIDIARY OF *Ford Motor Company*

Printed in U.S.A.